

# Vulkan® 1.1.129 - A Specification (with all registered Vulkan extensions)

The Khronos® Vulkan Working Group

Version 1.1.129, 2019-11-24 13:26:36Z

# Table of Contents

1. Introduction .....	3
1.1. Document Conventions .....	3
2. Fundamentals .....	5
2.1. Host and Device Environment .....	5
2.2. Execution Model .....	5
2.3. Object Model .....	7
2.4. Application Binary Interface .....	11
2.5. Command Syntax and Duration .....	12
2.6. Threading Behavior .....	13
2.7. Errors .....	24
2.8. Numeric Representation and Computation .....	44
2.9. Fixed-Point Data Conversions .....	46
2.10. Common Object Types .....	47
3. Initialization .....	49
3.1. Command Function Pointers .....	49
3.2. Instances .....	52
4. Devices and Queues .....	62
4.1. Physical Devices .....	62
4.2. Devices .....	82
4.3. Queues .....	94
5. Command Buffers .....	102
5.1. Command Buffer Lifecycle .....	102
5.2. Command Pools .....	104
5.3. Command Buffer Allocation and Management .....	110
5.4. Command Buffer Recording .....	114
5.5. Command Buffer Submission .....	121
5.6. Queue Forward Progress .....	135
5.7. Secondary Command Buffer Execution .....	136
5.8. Command Buffer Device Mask .....	139
6. Synchronization and Cache Control .....	142
6.1. Execution and Memory Dependencies .....	142
6.2. Implicit Synchronization Guarantees .....	159
6.3. Fences .....	161
6.4. Semaphores .....	185
6.5. Events .....	211
6.6. Pipeline Barriers .....	224
6.7. Memory Barriers .....	230
6.8. Wait Idle Operations .....	239

6.9. Host Write Ordering Guarantees . . . . .	240
6.10. Synchronization and Multiple Physical Devices . . . . .	241
6.11. Calibrated timestamps . . . . .	241
7. Render Pass . . . . .	245
7.1. Render Pass Creation . . . . .	246
7.2. Render Pass Compatibility . . . . .	295
7.3. Framebuffers . . . . .	295
7.4. Render Pass Commands . . . . .	305
8. Shaders . . . . .	326
8.1. Shader Modules . . . . .	326
8.2. Shader Execution . . . . .	330
8.3. Shader Memory Access Ordering . . . . .	330
8.4. Shader Inputs and Outputs . . . . .	331
8.5. Task Shaders . . . . .	331
8.6. Mesh Shaders . . . . .	332
8.7. Vertex Shaders . . . . .	332
8.8. Tessellation Control Shaders . . . . .	333
8.9. Tessellation Evaluation Shaders . . . . .	333
8.10. Geometry Shaders . . . . .	334
8.11. Fragment Shaders . . . . .	334
8.12. Compute Shaders . . . . .	336
8.13. Interpolation Decorations . . . . .	336
8.14. Ray Generation Shaders . . . . .	338
8.15. Intersection Shaders . . . . .	338
8.16. Any-Hit Shaders . . . . .	338
8.17. Closest Hit Shaders . . . . .	339
8.18. Miss Shaders . . . . .	339
8.19. Callable Shaders . . . . .	339
8.20. Static Use . . . . .	339
8.21. Invocation and Derivative Groups . . . . .	340
8.22. Subgroups . . . . .	340
8.23. Cooperative Matrices . . . . .	342
8.24. Validation Cache . . . . .	346
9. Pipelines . . . . .	353
9.1. Compute Pipelines . . . . .	354
9.2. Graphics Pipelines . . . . .	363
9.3. Pipeline destruction . . . . .	379
9.4. Multiple Pipeline Creation . . . . .	380
9.5. Pipeline Derivatives . . . . .	381
9.6. Pipeline Cache . . . . .	381
9.7. Specialization Constants . . . . .	388

9.8. Pipeline Binding .....	391
9.9. Dynamic State .....	394
9.10. Pipeline Shader Information .....	395
9.11. Pipeline Compiler Control .....	407
9.12. Ray Tracing Pipeline .....	407
9.13. Pipeline Creation Feedback .....	415
10. Memory Allocation .....	418
10.1. Host Memory .....	418
10.2. Device Memory .....	425
11. Resource Creation .....	489
11.1. Buffers .....	489
11.2. Buffer Views .....	498
11.3. Images .....	503
11.4. Image Layouts .....	536
11.5. Image Views .....	540
11.6. Resource Memory Association .....	558
11.7. Resource Sharing Mode .....	587
11.8. Memory Aliasing .....	589
11.9. Acceleration Structures .....	591
12. Samplers .....	606
12.1. Sampler Y'CbCr conversion .....	615
13. Resource Descriptors .....	625
13.1. Descriptor Types .....	625
13.2. Descriptor Sets .....	629
13.3. Physical Storage Buffer Access .....	699
14. Shader Interfaces .....	702
14.1. Shader Input and Output Interfaces .....	702
14.2. Vertex Input Interface .....	706
14.3. Fragment Output Interface .....	706
14.4. Fragment Input Attachment Interface .....	707
14.5. Shader Resource Interface .....	708
14.6. Built-In Variables .....	716
15. Image Operations .....	744
15.1. Image Operations Overview .....	744
15.2. Conversion Formulas .....	748
15.3. Texel Input Operations .....	749
15.4. Texel Output Operations .....	765
15.5. Derivative Operations .....	767
15.6. Normalized Texel Coordinate Operations .....	768
15.7. Unnormalized Texel Coordinate Operations .....	775
15.8. Integer Texel Coordinate Operations .....	776

15.9. Image Sample Operations .....	777
15.10. Texel Footprint Evaluation .....	781
15.11. Image Operation Steps .....	783
16. Fragment Density Map Operations .....	785
16.1. Fragment Density Map Operations Overview .....	785
16.2. Fetch Density Value .....	785
16.3. Fragment Area Conversion .....	786
17. Queries .....	788
17.1. Query Pools .....	788
17.2. Query Operation .....	793
17.3. Occlusion Queries .....	812
17.4. Pipeline Statistics Queries .....	813
17.5. Timestamp Queries .....	815
17.6. Performance Queries .....	818
17.7. Transform Feedback Queries .....	821
17.8. Intel performance queries .....	822
18. Clear Commands .....	835
18.1. Clearing Images Outside A Render Pass Instance .....	835
18.2. Clearing Images Inside A Render Pass Instance .....	840
18.3. Clear Values .....	844
18.4. Filling Buffers .....	846
18.5. Updating Buffers .....	847
19. Copy Commands .....	850
19.1. Common Operation .....	850
19.2. Copying Data Between Buffers .....	851
19.3. Copying Data Between Images .....	853
19.4. Copying Data Between Buffers and Images .....	861
19.5. Image Copies with Scaling .....	872
19.6. Resolving Multisample Images .....	881
19.7. Buffer Markers .....	885
20. Drawing Commands .....	888
20.1. Primitive Topologies .....	889
20.2. Primitive Order .....	895
20.3. Programmable Primitive Shading .....	896
20.4. Conditional Rendering .....	929
20.5. Programmable Mesh Shading .....	933
21. Fixed-Function Vertex Processing .....	945
21.1. Vertex Attributes .....	945
21.2. Vertex Input Description .....	950
21.3. Vertex Attribute Divisor in Instanced Rendering .....	955
21.4. Example .....	957

22. Tessellation .....	959
22.1. Tessellator .....	959
22.2. Tessellator Patch Discard .....	961
22.3. Tessellator Spacing .....	961
22.4. Tessellation Primitive Ordering .....	962
22.5. Tessellator Vertex Winding Order .....	962
22.6. Triangle Tessellation .....	963
22.7. Quad Tessellation .....	964
22.8. Isoline Tessellation .....	966
22.9. Tessellation Point Mode .....	966
22.10. Tessellation Pipeline State .....	967
23. Geometry Shading .....	970
23.1. Geometry Shader Input Primitives .....	970
23.2. Geometry Shader Output Primitives .....	971
23.3. Multiple Invocations of Geometry Shaders .....	971
23.4. Geometry Shader Primitive Ordering .....	971
23.5. Geometry Shader Passthrough .....	971
24. Mesh Shading .....	974
24.1. Task Shader Input .....	974
24.2. Task Shader Output .....	974
24.3. Mesh Generation .....	974
24.4. Mesh Shader Input .....	974
24.5. Mesh Shader Output Primitives .....	974
24.6. Mesh Shader Per-View Outputs .....	975
24.7. Mesh Shader Primitive Ordering .....	975
25. Fixed-Function Vertex Post-Processing .....	976
25.1. Transform Feedback .....	976
25.2. Viewport Swizzle .....	984
25.3. Flat Shading .....	986
25.4. Primitive Clipping .....	987
25.5. Clipping Shader Outputs .....	989
25.6. Controlling Viewport W Scaling .....	989
25.7. Coordinate Transformations .....	992
25.8. Controlling the Viewport .....	992
26. Rasterization .....	998
26.1. Discarding Primitives Before Rasterization .....	1003
26.2. Controlling the Vertex Stream Used for Rasterization .....	1003
26.3. Rasterization Order .....	1005
26.4. Multisampling .....	1006
26.5. Custom Sample Locations .....	1009
26.6. Shading Rate Image .....	1012

26.7. Sample Shading . . . . .	1025
26.8. Barycentric Interpolation . . . . .	1025
26.9. Points . . . . .	1027
26.10. Line Segments . . . . .	1028
26.11. Polygons . . . . .	1037
27. Fragment Operations . . . . .	1047
27.1. Early Per-Fragment Tests . . . . .	1047
27.2. Discard Rectangles Test . . . . .	1047
27.3. Scissor Test . . . . .	1051
27.4. Exclusive Scissor Test . . . . .	1053
27.5. Sample Mask . . . . .	1056
27.6. Early Fragment Test Mode . . . . .	1057
27.7. Late Per-Fragment Tests . . . . .	1057
27.8. Mixed attachment samples . . . . .	1057
27.9. Multisample Coverage . . . . .	1058
27.10. Depth and Stencil Operations . . . . .	1059
27.11. Depth Bounds Test . . . . .	1060
27.12. Stencil Test . . . . .	1061
27.13. Depth Test . . . . .	1068
27.14. Representative Fragment Test . . . . .	1069
27.15. Sample Counting . . . . .	1070
27.16. Fragment Coverage To Color . . . . .	1070
27.17. Coverage Reduction . . . . .	1071
28. The Framebuffer . . . . .	1078
28.1. Blending . . . . .	1078
28.2. Logical Operations . . . . .	1096
28.3. Color Write Mask . . . . .	1098
29. Dispatching Commands . . . . .	1100
30. Device-Generated Commands . . . . .	1111
30.1. Features and Limitations . . . . .	1111
30.2. Binding Object Table . . . . .	1113
30.3. Indirect Commands Layout . . . . .	1123
30.4. Indirect Commands Generation . . . . .	1132
31. Sparse Resources . . . . .	1140
31.1. Sparse Resource Features . . . . .	1140
31.2. Sparse Buffers and Fully-Resident Images . . . . .	1141
31.3. Sparse Partially-Resident Buffers . . . . .	1142
31.4. Sparse Partially-Resident Images . . . . .	1142
31.5. Sparse Memory Aliasing . . . . .	1150
31.6. Sparse Resource Implementation Guidelines . . . . .	1150
31.7. Sparse Resource API . . . . .	1152

31.8. Examples .....	1177
32. Window System Integration (WSI) .....	1183
32.1. WSI Platform .....	1183
32.2. WSI Surface .....	1183
32.3. Presenting Directly to Display Devices .....	1205
32.4. Querying for WSI Support .....	1231
32.5. Surface Queries .....	1235
32.6. Full Screen Exclusive Control .....	1259
32.7. Device Group Queries .....	1260
32.8. Display Timing Queries .....	1265
32.9. WSI Swapchain .....	1271
32.10. Hdr Metadata .....	1310
33. Ray Tracing .....	1312
33.1. Ray Tracing Commands .....	1312
33.2. Shader Binding Table .....	1318
33.3. Acceleration Structures .....	1320
34. Extending Vulkan .....	1329
34.1. Instance and Device Functionality .....	1329
34.2. Core Versions .....	1329
34.3. Layers .....	1331
34.4. Extensions .....	1335
34.5. Extension Dependencies .....	1338
34.6. Compatibility Guarantees (Informative) .....	1338
35. Features .....	1343
35.1. Feature Requirements .....	1390
36. Limits .....	1392
36.1. Limit Requirements .....	1436
36.2. Additional Multisampling Capabilities .....	1449
37. Formats .....	1451
37.1. Format Definition .....	1451
37.2. Format Properties .....	1492
37.3. Required Format Support .....	1500
38. Additional Capabilities .....	1517
38.1. Additional Image Capabilities .....	1517
38.2. Additional Buffer Capabilities .....	1539
38.3. Optional Semaphore Capabilities .....	1541
38.4. Optional Fence Capabilities .....	1547
38.5. Timestamp Calibration Capabilities .....	1551
39. Debugging .....	1553
39.1. Debug Utilities .....	1556
39.2. Debug Markers .....	1574

39.3. Debug Report Callbacks .....	1581
39.4. Device Loss Debugging .....	1589
Appendix A: Vulkan Environment for SPIR-V .....	1592
Versions and Formats .....	1592
Capabilities .....	1592
Validation Rules within a Module .....	1598
Precision and Operation of SPIR-V Instructions .....	1606
Compatibility Between SPIR-V Image Formats And Vulkan Formats .....	1611
Appendix B: Memory Model .....	1613
Agent .....	1613
Memory Location .....	1613
Allocation .....	1613
Memory Operation .....	1613
Reference .....	1614
Program-Order .....	1614
Scope .....	1615
Atomic Operation .....	1615
Scoped Modification Order .....	1616
Memory Semantics .....	1616
Release Sequence .....	1618
Synchronizes-With .....	1618
System-Synchronizes-With .....	1620
Private vs. Non-Private .....	1620
Inter-Thread-Happens-Before .....	1620
Happens-Before .....	1621
Availability and Visibility .....	1621
Availability, Visibility, and Domain Operations .....	1623
Availability and Visibility Semantics .....	1624
Per-Instruction Availability and Visibility Semantics .....	1624
Location-Ordered .....	1625
Data Race .....	1626
Visible-To .....	1626
Acyclicity .....	1626
Shader I/O .....	1627
Deallocation .....	1627
Informative Descriptions .....	1628
Tessellation Output Ordering .....	1628
Cooperative Matrix Memory Access .....	1629
Appendix C: Compressed Image Formats .....	1630
Block-Compressed Image Formats .....	1631
ETC Compressed Image Formats .....	1632

ASTC Compressed Image Formats .....	1633
PVRTC Compressed Image Formats .....	1635
Appendix D: Core Revisions (Informative) .....	1636
Version 1.1 .....	1636
Appendix E: Layers & Extensions (Informative) .....	1646
List of Current Extensions .....	1646
List of Deprecated Extensions .....	2001
Appendix F: API Boilerplate .....	2096
Vulkan Header Files .....	2096
Window System-Specific Header Control (Informative) .....	2098
Appendix G: Invariance .....	2101
Repeatability .....	2101
Multi-pass Algorithms .....	2101
Invariance Rules .....	2101
Tessellation Invariance .....	2103
Glossary .....	2105
Common Abbreviations .....	2130
Prefixes .....	2132
Appendix H: Credits (Informative) .....	2133
Working Group Contributors to Vulkan 1.1 and 1.0 .....	2133
Working Group Contributors to Vulkan 1.1 .....	2135
Working Group Contributors to Vulkan 1.0 .....	2137
Other Credits .....	2139

## Copyright 2014-2019 The Khronos Group Inc.

This Specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos. Khronos grants a conditional copyright license to use and reproduce the unmodified Specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms.

Khronos makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this Specification, including, without limitation: merchantability, fitness for a particular purpose, non-infringement of any intellectual property, correctness, accuracy, completeness, timeliness, and reliability. Under no circumstances will Khronos, or any of its Promoters, Contributors or Members, or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

This Specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at [https://www.khronos.org/files/member\\_agreement.pdf](https://www.khronos.org/files/member_agreement.pdf), and which defines the terms 'Scope', 'Compliant Portion', and 'Necessary Patent Claims'. Parties desiring to implement the Specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos Intellectual Property Rights Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this Specification; see <https://www.khronos.org/adopters>.

This Specification contains substantially unmodified functionality from, and is a successor to, Khronos specifications including OpenGL, OpenGL ES and OpenCL.

Some parts of this Specification are purely informative and so are EXCLUDED from the Scope of this Specification. The [Document Conventions](#) section of the [Introduction](#) defines how these parts of the Specification are identified.

Where this Specification uses [technical terminology](#), defined in the [Glossary](#) or otherwise, that refer to enabling technologies that are not expressly set forth in this Specification, those enabling technologies are EXCLUDED from the Scope of this Specification. For clarity, enabling technologies not disclosed with particularity in this Specification (e.g. semiconductor manufacturing technology, hardware architecture, processor architecture or microarchitecture, memory architecture, compiler technology, object oriented technology, basic operating system technology, compression technology, algorithms, and so on) are NOT to be considered expressly set forth; only those application program interfaces and data structures disclosed with particularity are included in the Scope of this Specification.

For purposes of the Khronos Intellectual Property Rights Policy as it relates to the definition of Necessary Patent Claims, all recommended or optional features, behaviors and functionality set forth in this Specification, if implemented, are considered to be included as Compliant Portions.

Where this Specification includes [normative references to external documents](#), only the specifically identified sections of those external documents are INCLUDED in the Scope of this Specification. If not created by Khronos, those external documents may contain contributions from non-members of Khronos not covered by the Khronos Intellectual Property Rights Policy.

This document contains extensions which are not ratified by Khronos, and as such is not a ratified Specification, though it contains text from (and is a superset of) the ratified Vulkan Specification. The ratified versions of the Vulkan Specification can be found at <https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html> (core only) and <https://www.khronos.org/registry/vulkan/specs/1.1-khr-extensions/html/vkspec.html> (core with KHR extensions).

Vulkan and Khronos are registered trademarks of The Khronos Group Inc. ASTC is a trademark of ARM Holdings PLC; OpenCL is a trademark of Apple Inc.; and OpenGL and OpenGL ES are registered trademarks of Hewlett Packard Enterprise, all used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

# Chapter 1. Introduction

This document, referred to as the “Vulkan Specification” or just the “Specification” hereafter, describes the Vulkan Application Programming Interface (API). Vulkan is a C99 API designed for explicit control of low-level graphics and compute functionality.

The canonical version of the Specification is available in the official [Vulkan Registry](https://www.khronos.org/registry/vulkan/) (<https://www.khronos.org/registry/vulkan/>). The source files used to generate the Vulkan specification are stored in the [Vulkan Documentation Repository](https://github.com/KhronosGroup/Vulkan-Docs) (<https://github.com/KhronosGroup/Vulkan-Docs>). The source repository additionally has a public issue tracker and allows the submission of pull requests that improve the specification.

## 1.1. Document Conventions

The Vulkan specification is intended for use by both implementors of the API and application developers seeking to make use of the API, forming a contract between these parties. Specification text may address either party; typically the intended audience can be inferred from context, though some sections are defined to address only one of these parties. (For example, [Valid Usage](#) sections only address application developers). Any requirements, prohibitions, recommendations or options defined by [normative terminology](#) are imposed only on the audience of that text.

*Note*



Structure and enumerated types defined in extensions that were promoted to core in Vulkan 1.1 are now defined in terms of the equivalent Vulkan 1.1 interfaces. This affects the Vulkan Specification, the Vulkan header files, and the corresponding XML Registry.

### 1.1.1. Normative Terminology

Within this specification, the key words **must**, **required**, **should**, **recommended**, **may**, and **optional** are to be interpreted as described in [RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels](https://www.ietf.org/rfc/rfc2119.txt) (<https://www.ietf.org/rfc/rfc2119.txt>). These key words are highlighted in the specification for clarity. In text addressing application developers, their use expresses requirements that apply to application behavior. In text addressing implementors, their use expresses requirements that apply to implementations.

In text addressing application developers, the additional key words **can** and **cannot** are to be interpreted as describing the capabilities of an application, as follows:

#### **can**

This word means that the application is able to perform the action described.

#### **cannot**

This word means that the API and/or the execution environment provide no mechanism through which the application can express or accomplish the action described.

These key words are never used in text addressing implementors.

#### Note



There is an important distinction between **cannot** and **must not**, as used in this Specification. **Cannot** means something the application literally is unable to express or accomplish through the API, while **must not** means something that the application is capable of expressing through the API, but that the consequences of doing so are undefined and potentially unrecoverable for the implementation (see [Errors](#)).

Unless otherwise noted in the section heading, all sections and appendices in this document are normative.

### 1.1.2. Technical Terminology

The Vulkan Specification makes use of common engineering and graphics terms such as **Pipeline**, **Shader**, and **Host** to identify and describe Vulkan API constructs and their attributes, states, and behaviors. The [Glossary](#) defines the basic meanings of these terms in the context of the Specification. The Specification text provides fuller definitions of the terms and may elaborate, extend, or clarify the [Glossary](#) definitions. When a term defined in the [Glossary](#) is used in normative language within the Specification, the definitions within the Specification govern and supersede any meanings the terms may have in other technical contexts (i.e. outside the Specification).

### 1.1.3. Normative References

References to external documents are considered normative references if the Specification uses any of the normative terms defined in [Normative Terminology](#) to refer to them or their requirements, either as a whole or in part.

The following documents are referenced by normative sections of the specification:

IEEE. August, 2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008. <https://dx.doi.org/10.1109/IEEESTD.2008.4610935>.

Andrew Garrard. *Khronos Data Format Specification, version 1.2* (March, 2019). <https://www.khronos.org/registry/DataFormat/specs/1.2/dataformat.1.2.html>.

John Kessenich. *SPIR-V Extended Instructions for GLSL, Version 1.00* (February 10, 2016). <https://www.khronos.org/registry/spir-v/>.

John Kessenich, Boaz Ouriel, and Raun Krisch. *SPIR-V Specification, Version 1.4, Revision 1, Unified* (December 14, 2018). <https://www.khronos.org/registry/spir-v/>.

Jon Leech and Tobias Hector. *Vulkan Documentation and Extensions: Procedures and Conventions* (July 20, 2019). <https://www.khronos.org/registry/vulkan/specs/1.1/styleguide.html>.

*Vulkan Loader Specification and Architecture Overview* (August, 2016). <https://github.com/KhronosGroup/Vulkan-Loader/blob/master/loader/LoaderAndLayerInterface.md>.

# Chapter 2. Fundamentals

This chapter introduces fundamental concepts including the Vulkan architecture and execution model, API syntax, queues, pipeline configurations, numeric representation, state and state queries, and the different types of objects and shaders. It provides a framework for interpreting more specific descriptions of commands and behavior in the remainder of the Specification.

## 2.1. Host and Device Environment

The Vulkan Specification assumes and requires: the following properties of the host environment with respect to Vulkan implementations:

- The host **must** have runtime support for 8, 16, 32 and 64-bit signed and unsigned two-complement integers, all addressable at the granularity of their size in bytes.
- The host **must** have runtime support for 32- and 64-bit floating-point types satisfying the range and precision constraints in the [Floating Point Computation](#) section.
- The representation and endianness of these types on the host **must** match the representation and endianness of the same types on every physical device supported.

*Note*



Since a variety of data types and structures in Vulkan **may** be accessible by both host and physical device operations, the implementation **should** be able to access such data efficiently in both paths in order to facilitate writing portable and performant applications.

## 2.2. Execution Model

This section outlines the execution model of a Vulkan system.

Vulkan exposes one or more *devices*, each of which exposes one or more *queues* which **may** process work asynchronously to one another. The set of queues supported by a device is partitioned into *families*. Each family supports one or more types of functionality and **may** contain multiple queues with similar characteristics. Queues within a single family are considered *compatible* with one another, and work produced for a family of queues **can** be executed on any queue within that family. This Specification defines four types of functionality that queues **may** support: graphics, compute, transfer, and sparse memory management.

*Note*



A single device **may** report multiple similar queue families rather than, or as well as, reporting multiple members of one or more of those families. This indicates that while members of those families have similar capabilities, they are *not* directly compatible with one another.

Device memory is explicitly managed by the application. Each device **may** advertise one or more heaps, representing different areas of memory. Memory heaps are either device local or host local,

but are always visible to the device. Further detail about memory heaps is exposed via memory types available on that heap. Examples of memory areas that **may** be available on an implementation include:

- *device local* is memory that is physically connected to the device.
- *device local, host visible* is device local memory that is visible to the host.
- *host local, host visible* is memory that is local to the host and visible to the device and host.

On other architectures, there **may** only be a single heap that **can** be used for any purpose.

A Vulkan application controls a set of devices through the submission of command buffers which have recorded device commands issued via Vulkan library calls. The content of command buffers is specific to the underlying implementation and is opaque to the application. Once constructed, a command buffer **can** be submitted once or many times to a queue for execution. Multiple command buffers **can** be built in parallel by employing multiple threads within the application.

Command buffers submitted to different queues **may** execute in parallel or even out of order with respect to one another. Command buffers submitted to a single queue respect [submission order](#), as described further in [synchronization chapter](#). Command buffer execution by the device is also asynchronous to host execution. Once a command buffer is submitted to a queue, control **may** return to the application immediately. Synchronization between the device and host, and between different queues is the responsibility of the application.

### 2.2.1. Queue Operation

Vulkan queues provide an interface to the execution engines of a device. Commands for these execution engines are recorded into command buffers ahead of execution time. These command buffers are then submitted to queues with a *queue submission* command for execution in a number of *batches*. Once submitted to a queue, these commands will begin and complete execution without further application intervention, though the order of this execution is dependent on a number of [implicit and explicit ordering constraints](#).

Work is submitted to queues using queue submission commands that typically take the form `vkQueue*` (e.g. `vkQueueSubmit`, `vkQueueBindSparse`), and optionally take a list of semaphores upon which to wait before work begins and a list of semaphores to signal once work has completed. The work itself, as well as signaling and waiting on the semaphores are all *queue operations*.

Queue operations on different queues have no implicit ordering constraints, and **may** execute in any order. Explicit ordering constraints between queues **can** be expressed with [semaphores](#) and [fences](#).

Command buffer submissions to a single queue respect [submission order](#) and other [implicit ordering guarantees](#), but otherwise **may** overlap or execute out of order. Other types of batches and queue submissions against a single queue (e.g. [sparse memory binding](#)) have no implicit ordering constraints with any other queue submission or batch. Additional explicit ordering constraints between queue submissions and individual batches can be expressed with [semaphores](#) and [fences](#).

Before a fence or semaphore is signaled, it is guaranteed that any previously submitted queue operations have completed execution, and that memory writes from those queue operations are

[available](#) to future queue operations. Waiting on a signaled semaphore or fence guarantees that previous writes that are available are also [visible](#) to subsequent commands.

Command buffer boundaries, both between primary command buffers of the same or different batches or submissions as well as between primary and secondary command buffers, do not introduce any additional ordering constraints. In other words, submitting the set of command buffers (which **can** include executing secondary command buffers) between any semaphore or fence operations execute the recorded commands as if they had all been recorded into a single primary command buffer, except that the current state is [reset](#) on each boundary. Explicit ordering constraints **can** be expressed with [explicit synchronization primitives](#).

There are a few [implicit ordering guarantees](#) between commands within a command buffer, but only covering a subset of execution. Additional explicit ordering constraints can be expressed with the various [explicit synchronization primitives](#).

*Note*



Implementations have significant freedom to overlap execution of work submitted to a queue, and this is common due to deep pipelining and parallelism in Vulkan devices.

Commands recorded in command buffers either perform actions (draw, dispatch, clear, copy, query/timestamp operations, begin/end subpass operations), set state (bind pipelines, descriptor sets, and buffers, set dynamic state, push constants, set render pass/subpass state), or perform synchronization (set/wait events, pipeline barrier, render pass/subpass dependencies). Some commands perform more than one of these tasks. State setting commands update the *current state* of the command buffer. Some commands that perform actions (e.g. draw/dispatch) do so based on the current state set cumulatively since the start of the command buffer. The work involved in performing action commands is often allowed to overlap or to be reordered, but doing so **must** not alter the state to be used by each action command. In general, action commands are those commands that alter framebuffer attachments, read/write buffer or image memory, or write to query pools.

Synchronization commands introduce explicit [execution and memory dependencies](#) between two sets of action commands, where the second set of commands depends on the first set of commands. These dependencies enforce that both the execution of certain [pipeline stages](#) in the later set occur after the execution of certain stages in the source set, and that the effects of [memory accesses](#) performed by certain pipeline stages occur in order and are visible to each other. When not enforced by an explicit dependency or [implicit ordering guarantees](#), action commands **may** overlap execution or execute out of order, and **may** not see the side effects of each other's memory accesses.

The device executes queue operations asynchronously with respect to the host. Control is returned to an application immediately following command buffer submission to a queue. The application **must** synchronize work between the host and device as needed.

## 2.3. Object Model

The devices, queues, and other entities in Vulkan are represented by Vulkan objects. At the API

level, all objects are referred to by handles. There are two classes of handles, dispatchable and non-dispatchable. *Dispatchable* handle types are a pointer to an opaque type. This pointer **may** be used by layers as part of intercepting API commands, and thus each API command takes a dispatchable type as its first parameter. Each object of a dispatchable type **must** have a unique handle value during its lifetime.

*Non-dispatchable* handle types are a 64-bit integer type whose meaning is implementation-dependent, and **may** encode object information directly in the handle rather than acting as a reference to an underlying object. Objects of a non-dispatchable type **may** not have unique handle values within a type or across types. If handle values are not unique, then destroying one such handle **must** not cause identical handles of other types to become invalid, and **must** not cause identical handles of the same type to become invalid if that handle value has been created more times than it has been destroyed.

All objects created or allocated from a `VkDevice` (i.e. with a `VkDevice` as the first parameter) are private to that device, and **must** not be used on other devices.

### 2.3.1. Object Lifetime

Objects are created or allocated by `vkCreate*` and `vkAllocate*` commands, respectively. Once an object is created or allocated, its “structure” is considered to be immutable, though the contents of certain object types is still free to change. Objects are destroyed or freed by `vkDestroy*` and `vkFree*` commands, respectively.

Objects that are allocated (rather than created) take resources from an existing pool object or memory heap, and when freed return resources to that pool or heap. While object creation and destruction are generally expected to be low-frequency occurrences during runtime, allocating and freeing objects **can** occur at high frequency. Pool objects help accommodate improved performance of the allocations and frees.

It is an application’s responsibility to track the lifetime of Vulkan objects, and not to destroy them while they are still in use.

The ownership of application-owned memory is immediately acquired by any Vulkan command it is passed into. Ownership of such memory **must** be released back to the application at the end of the duration of the command, so that the application **can** alter or free this memory as soon as all the commands that acquired it have returned.

The following object types are consumed when they are passed into a Vulkan command and not further accessed by the objects they are used to create. They **must** not be destroyed in the duration of any API command they are passed into:

- `VkShaderModule`
- `VkPipelineCache`
- `VkValidationCacheEXT`

A `VkRenderPass` object passed as a parameter to create another object is not further accessed by that object after the duration of the command it is passed into. A `VkRenderPass` used in a command buffer follows the rules described below.

A `VkPipelineLayout` object **must** not be destroyed while any command buffer that uses it is in the recording state.

`VkDescriptorSetLayout` objects **may** be accessed by commands that operate on descriptor sets allocated using that layout, and those descriptor sets **must** not be updated with `vkUpdateDescriptorSets` after the descriptor set layout has been destroyed. Otherwise, a `VkDescriptorSetLayout` object passed as a parameter to create another object is not further accessed by that object after the duration of the command it is passed into.

The application **must** not destroy any other type of Vulkan object until all uses of that object by the device (such as via command buffer execution) have completed.

The following Vulkan objects **must** not be destroyed while any command buffers using the object are in the [pending state](#):

- `VkEvent`
- `VkQueryPool`
- `VkBuffer`
- `VkBufferView`
- `VkImage`
- `VkImageView`
- `VkPipeline`
- `VkSampler`
- `VkSamplerYcbcrConversion`
- `VkDescriptorPool`
- `VkFramebuffer`
- `VkRenderPass`
- `VkCommandBuffer`
- `VkCommandPool`
- `VkDeviceMemory`
- `VkDescriptorSet`
- `VkObjectTableNVX`
- `VkIndirectCommandsLayoutNVX`

Destroying these objects will move any command buffers that are in the [recording or executable state](#), and are using those objects, to the [invalid state](#).

The following Vulkan objects **must** not be destroyed while any queue is executing commands that use the object:

- `VkFence`
- `VkSemaphore`
- `VkCommandBuffer`
- `VkCommandPool`

In general, objects **can** be destroyed or freed in any order, even if the object being freed is involved in the use of another object (e.g. use of a resource in a view, use of a view in a descriptor set, use of

an object in a command buffer, binding of a memory allocation to a resource), as long as any object that uses the freed object is not further used in any way except to be destroyed or to be reset in such a way that it no longer uses the other object (such as resetting a command buffer). If the object has been reset, then it **can** be used as if it never used the freed object. An exception to this is when there is a parent/child relationship between objects. In this case, the application **must** not destroy a parent object before its children, except when the parent is explicitly defined to free its children when it is destroyed (e.g. for pool objects, as defined below).

`VkCommandPool` objects are parents of `VkCommandBuffer` objects. `VkDescriptorPool` objects are parents of `VkDescriptorSet` objects. `VkDevice` objects are parents of many object types (all that take a `VkDevice` as a parameter to their creation).

The following Vulkan objects have specific restrictions for when they **can** be destroyed:

- `VkQueue` objects **cannot** be explicitly destroyed. Instead, they are implicitly destroyed when the `VkDevice` object they are retrieved from is destroyed.
- Destroying a pool object implicitly frees all objects allocated from that pool. Specifically, destroying `VkCommandPool` frees all `VkCommandBuffer` objects that were allocated from it, and destroying `VkDescriptorPool` frees all `VkDescriptorSet` objects that were allocated from it.
- `VkDevice` objects **can** be destroyed when all `VkQueue` objects retrieved from them are idle, and all objects created from them have been destroyed. This includes the following objects:
  - `VkFence`
  - `VkSemaphore`
  - `VkEvent`
  - `VkQueryPool`
  - `VkBuffer`
  - `VkBufferView`
  - `VkImage`
  - `VkImageView`
  - `VkShaderModule`
  - `VkPipelineCache`
  - `VkPipeline`
  - `VkPipelineLayout`
  - `VkSampler`
  - `VkSamplerYcbcrConversion`
  - `VkDescriptorSetLayout`
  - `VkDescriptorPool`
  - `VkFramebuffer`
  - `VkRenderPass`
  - `VkCommandPool`
  - `VkCommandBuffer`
  - `VkDeviceMemory`
  - `VkValidationCacheEXT`
- `VkPhysicalDevice` objects **cannot** be explicitly destroyed. Instead, they are implicitly destroyed

when the `VkInstance` object they are retrieved from is destroyed.

- `VkInstance` objects **can** be destroyed once all `VkDevice` objects created from any of its `VkPhysicalDevice` objects have been destroyed.

### 2.3.2. External Object Handles

As defined above, the scope of object handles created or allocated from a `VkDevice` is limited to that logical device. Objects which are not in scope are said to be external. To bring an external object into scope, an external handle **must** be exported from the object in the source scope and imported into the destination scope.

*Note*



The scope of external handles and their associated resources **may** vary according to their type, but they **can** generally be shared across process and API boundaries.

## 2.4. Application Binary Interface

The mechanism by which Vulkan is made available to applications is platform- or implementation-defined. On many platforms the C interface described in this Specification is provided by a shared library. Since shared libraries can be changed independently of the applications that use them, they present particular compatibility challenges, and this Specification places some requirements on them.

Shared library implementations **must** use the default Application Binary Interface (ABI) of the standard C compiler for the platform, or provide customized API headers that cause application code to use the implementation’s non-default ABI. An ABI in this context means the size, alignment, and layout of C data types; the procedure calling convention; and the naming convention for shared library symbols corresponding to C functions. Customizing the calling convention for a platform is usually accomplished by defining [calling convention macros](#) appropriately in `vk_platform.h`.

On platforms where Vulkan is provided as a shared library, library symbols beginning with “vk” and followed by a digit or uppercase letter are reserved for use by the implementation. Applications which use Vulkan **must** not provide definitions of these symbols. This allows the Vulkan shared library to be updated with additional symbols for new API versions or extensions without causing symbol conflicts with existing applications.

Shared library implementations **should** provide library symbols for commands in the highest version of this Specification they support, and for [Window System Integration](#) extensions relevant to the platform. They **may** also provide library symbols for commands defined by additional extensions.

### Note



These requirements and recommendations are intended to allow implementors to take advantage of platform-specific conventions for SDKs, ABIs, library versioning mechanisms, etc. while still minimizing the code changes necessary to port applications or libraries between platforms. Platform vendors, or providers of the *de facto* standard Vulkan shared library for a platform, are encouraged to document what symbols the shared library provides and how it will be versioned when new symbols are added.

Applications **should** only rely on shared library symbols for commands in the minimum core version required by the application. `vkGetInstanceProcAddr` and `vkGetDeviceProcAddr` **should** be used to obtain function pointers for commands in core versions beyond the application's minimum required version.

## 2.5. Command Syntax and Duration

The Specification describes Vulkan commands as functions or procedures using C99 syntax. Language bindings for other languages such as C++ and JavaScript **may** allow for stricter parameter passing, or object-oriented interfaces.

Vulkan uses the standard C types for the base type of scalar parameters (e.g. types from `<stdint.h>`), with exceptions described below, or elsewhere in the text when appropriate:

`VkBool32` represents boolean `True` and `False` values, since C does not have a sufficiently portable built-in boolean type:

```
typedef uint32_t VkBool32;
```

`VK_TRUE` represents a boolean `True` (integer 1) value, and `VK_FALSE` a boolean `False` (integer 0) value.

All values returned from a Vulkan implementation in a `VkBool32` will be either `VK_TRUE` or `VK_FALSE`.

Applications **must** not pass any other values than `VK_TRUE` or `VK_FALSE` into a Vulkan implementation where a `VkBool32` is expected.

`VkDeviceSize` represents device memory size and offset values:

```
typedef uint64_t VkDeviceSize;
```

`VkDeviceAddress` represents device buffer address values:

```
typedef uint64_t VkDeviceAddress;
```

Commands that create Vulkan objects are of the form `vkCreate*` and take `Vk*CreateInfo` structures with the parameters needed to create the object. These Vulkan objects are destroyed with

commands of the form `vkDestroy*`.

The last in-parameter to each command that creates or destroys a Vulkan object is `pAllocator`. The `pAllocator` parameter **can** be set to a non-`NULL` value such that allocations for the given object are delegated to an application provided callback; refer to the [Memory Allocation](#) chapter for further details.

Commands that allocate Vulkan objects owned by pool objects are of the form `vkAllocate*`, and take `Vk*AllocateInfo` structures. These Vulkan objects are freed with commands of the form `vkFree*`. These objects do not take allocators; if host memory is needed, they will use the allocator that was specified when their parent pool was created.

Commands are recorded into a command buffer by calling API commands of the form `vkCmd*`. Each such command **may** have different restrictions on where it **can** be used: in a primary and/or secondary command buffer, inside and/or outside a render pass, and in one or more of the supported queue types. These restrictions are documented together with the definition of each such command.

The *duration* of a Vulkan command refers to the interval between calling the command and its return to the caller.

### 2.5.1. Lifetime of Retrieved Results

Information is retrieved from the implementation with commands of the form `vkGet*` and `vkEnumerate*`.

Unless otherwise specified for an individual command, the results are *invariant*; that is, they will remain unchanged when retrieved again by calling the same command with the same parameters, so long as those parameters themselves all remain valid.

## 2.6. Threading Behavior

Vulkan is intended to provide scalable performance when used on multiple host threads. All commands support being called concurrently from multiple threads, but certain parameters, or components of parameters are defined to be *externally synchronized*. This means that the caller **must** guarantee that no more than one thread is using such a parameter at a given time.

More precisely, Vulkan commands use simple stores to update the state of Vulkan objects. A parameter declared as externally synchronized **may** have its contents updated at any time during the host execution of the command. If two commands operate on the same object and at least one of the commands declares the object to be externally synchronized, then the caller **must** guarantee not only that the commands do not execute simultaneously, but also that the two commands are separated by an appropriate memory barrier (if needed).

*Note*



Memory barriers are particularly relevant for hosts based on the ARM CPU architecture, which is more weakly ordered than many developers are accustomed to from x86/x64 programming. Fortunately, most higher-level synchronization primitives (like the pthread library) perform memory barriers as a part of mutual exclusion, so mutexing Vulkan objects via these primitives will have the desired effect.

Similarly the application **must** avoid any potential data hazard of application-owned memory that has its [ownership temporarily acquired](#) by a Vulkan command. While the ownership of application-owned memory remains acquired by a command the implementation **may** read the memory at any point, and it **may** write non-**const** qualified memory at any point. Parameters referring to non-**const** qualified application-owned memory are not marked explicitly as *externally synchronized* in the Specification.

Many object types are *immutable*, meaning the objects **cannot** change once they have been created. These types of objects never need external synchronization, except that they **must** not be destroyed while they are in use on another thread. In certain special cases mutable object parameters are internally synchronized, making external synchronization unnecessary. One example of this is the use of a `VkPipelineCache` in `vkCreateGraphicsPipelines` and `vkCreateComputePipelines`, where external synchronization around such a heavyweight command would be impractical. The implementation **must** internally synchronize the cache in this example, and **may** be able to do so in the form of a much finer-grained mutex around the command. Any command parameters that are not labeled as externally synchronized are either not mutated by the command or are internally synchronized. Additionally, certain objects related to a command's parameters (e.g. command pools and descriptor pools) **may** be affected by a command, and **must** also be externally synchronized. These implicit parameters are documented as described below.

Parameters of commands that are externally synchronized are listed below.

## Externally Synchronized Parameters

- The `instance` parameter in `vkDestroyInstance`
- The `device` parameter in `vkDestroyDevice`
- The `queue` parameter in `vkQueueSubmit`
- The `fence` parameter in `vkQueueSubmit`
- The `queue` parameter in `vkQueueWaitIdle`
- The `memory` parameter in `vkFreeMemory`
- The `memory` parameter in `vkMapMemory`
- The `memory` parameter in `vkUnmapMemory`
- The `buffer` parameter in `vkBindBufferMemory`
- The `image` parameter in `vkBindImageMemory`
- The `queue` parameter in `vkQueueBindSparse`
- The `fence` parameter in `vkQueueBindSparse`
- The `fence` parameter in `vkDestroyFence`
- The `semaphore` parameter in `vkDestroySemaphore`
- The `event` parameter in `vkDestroyEvent`
- The `event` parameter in `vkSetEvent`
- The `event` parameter in `vkResetEvent`
- The `queryPool` parameter in `vkDestroyQueryPool`
- The `buffer` parameter in `vkDestroyBuffer`
- The `bufferView` parameter in `vkDestroyBufferView`
- The `image` parameter in `vkDestroyImage`
- The `imageView` parameter in `vkDestroyImageView`
- The `shaderModule` parameter in `vkDestroyShaderModule`
- The `pipelineCache` parameter in `vkDestroyPipelineCache`
- The `dstCache` parameter in `vkMergePipelineCaches`
- The `pipeline` parameter in `vkDestroyPipeline`
- The `pipelineLayout` parameter in `vkDestroyPipelineLayout`
- The `sampler` parameter in `vkDestroySampler`
- The `descriptorsetLayout` parameter in `vkDestroyDescriptorSetLayout`
- The `descriptorPool` parameter in `vkDestroyDescriptorPool`
- The `descriptorPool` parameter in `vkResetDescriptorPool`
- The `descriptorPool` member of the `pAllocateInfo` parameter in `vkAllocateDescriptorSets`
- The `descriptorPool` parameter in `vkFreeDescriptorSets`

- The `framebuffer` parameter in `vkDestroyFramebuffer`
- The `renderPass` parameter in `vkDestroyRenderPass`
- The `commandPool` parameter in `vkDestroyCommandPool`
- The `commandPool` parameter in `vkResetCommandPool`
- The `commandPool` member of the `pAllocateInfo` parameter in `vkAllocateCommandBuffers`
- The `commandPool` parameter in `vkFreeCommandBuffers`
- The `commandBuffer` parameter in `vkBeginCommandBuffer`
- The `commandBuffer` parameter in `vkEndCommandBuffer`
- The `commandBuffer` parameter in `vkResetCommandBuffer`
- The `commandBuffer` parameter in `vkCmdBindPipeline`
- The `commandBuffer` parameter in `vkCmdSetViewport`
- The `commandBuffer` parameter in `vkCmdSetScissor`
- The `commandBuffer` parameter in `vkCmdSetLineWidth`
- The `commandBuffer` parameter in `vkCmdSetDepthBias`
- The `commandBuffer` parameter in `vkCmdSetBlendConstants`
- The `commandBuffer` parameter in `vkCmdSetDepthBounds`
- The `commandBuffer` parameter in `vkCmdSetStencilCompareMask`
- The `commandBuffer` parameter in `vkCmdSetStencilWriteMask`
- The `commandBuffer` parameter in `vkCmdSetStencilReference`
- The `commandBuffer` parameter in `vkCmdBindDescriptorSets`
- The `commandBuffer` parameter in `vkCmdBindIndexBuffer`
- The `commandBuffer` parameter in `vkCmdBindVertexBuffers`
- The `commandBuffer` parameter in `vkCmdDraw`
- The `commandBuffer` parameter in `vkCmdDrawIndexed`
- The `commandBuffer` parameter in `vkCmdDrawIndirect`
- The `commandBuffer` parameter in `vkCmdDrawIndexedIndirect`
- The `commandBuffer` parameter in `vkCmdDispatch`
- The `commandBuffer` parameter in `vkCmdDispatchIndirect`
- The `commandBuffer` parameter in `vkCmdCopyBuffer`
- The `commandBuffer` parameter in `vkCmdCopyImage`
- The `commandBuffer` parameter in `vkCmdBlitImage`
- The `commandBuffer` parameter in `vkCmdCopyBufferToImage`
- The `commandBuffer` parameter in `vkCmdCopyImageToBuffer`
- The `commandBuffer` parameter in `vkCmdUpdateBuffer`
- The `commandBuffer` parameter in `vkCmdFillBuffer`

- The `commandBuffer` parameter in `vkCmdClearColorImage`
- The `commandBuffer` parameter in `vkCmdClearDepthStencilImage`
- The `commandBuffer` parameter in `vkCmdClearAttachments`
- The `commandBuffer` parameter in `vkCmdResolveImage`
- The `commandBuffer` parameter in `vkCmdSetEvent`
- The `commandBuffer` parameter in `vkCmdResetEvent`
- The `commandBuffer` parameter in `vkCmdWaitEvents`
- The `commandBuffer` parameter in `vkCmdPipelineBarrier`
- The `commandBuffer` parameter in `vkCmdBeginQuery`
- The `commandBuffer` parameter in `vkCmdEndQuery`
- The `commandBuffer` parameter in `vkCmdResetQueryPool`
- The `commandBuffer` parameter in `vkCmdWriteTimestamp`
- The `commandBuffer` parameter in `vkCmdCopyQueryPoolResults`
- The `commandBuffer` parameter in `vkCmdPushConstants`
- The `commandBuffer` parameter in `vkCmdBeginRenderPass`
- The `commandBuffer` parameter in `vkCmdNextSubpass`
- The `commandBuffer` parameter in `vkCmdEndRenderPass`
- The `commandBuffer` parameter in `vkCmdExecuteCommands`
- The `commandBuffer` parameter in `vkCmdSetDeviceMask`
- The `commandBuffer` parameter in `vkCmdDispatchBase`
- The `commandPool` parameter in `vkTrimCommandPool`
- The `ycbcrConversion` parameter in `vkDestroySamplerYcbcrConversion`
- The `descriptorUpdateTemplate` parameter in `vkDestroyDescriptorUpdateTemplate`
- The `descriptorSet` parameter in `vkUpdateDescriptorSetWithTemplate`
- The `surface` parameter in `vkDestroySurfaceKHR`
- The `surface` member of the `pCreateInfo` parameter in `vkCreateSwapchainKHR`
- The `oldSwapchain` member of the `pCreateInfo` parameter in `vkCreateSwapchainKHR`
- The `swapchain` parameter in `vkDestroySwapchainKHR`
- The `swapchain` parameter in `vkAcquireNextImageKHR`
- The `semaphore` parameter in `vkAcquireNextImageKHR`
- The `fence` parameter in `vkAcquireNextImageKHR`
- The `queue` parameter in `vkQueuePresentKHR`
- The `surface` parameter in `vkGetDeviceGroupSurfacePresentModesKHR`
- The `surface` parameter in `vkGetPhysicalDevicePresentRectanglesKHR`
- The `display` parameter in `vkCreateDisplayModeKHR`

- The `mode` parameter in `vkGetDisplayPlaneCapabilitiesKHR`
- The `commandBuffer` parameter in `vkCmdSetDeviceMaskKHR`
- The `commandBuffer` parameter in `vkCmdDispatchBaseKHR`
- The `commandPool` parameter in `vkTrimCommandPoolKHR`
- The `commandBuffer` parameter in `vkCmdPushDescriptorSetKHR`
- The `commandBuffer` parameter in `vkCmdPushDescriptorSetWithTemplateKHR`
- The `descriptorUpdateTemplate` parameter in `vkDestroyDescriptorUpdateTemplateKHR`
- The `descriptorSet` parameter in `vkUpdateDescriptorSetWithTemplateKHR`
- The `commandBuffer` parameter in `vkCmdBeginRenderPass2KHR`
- The `commandBuffer` parameter in `vkCmdNextSubpass2KHR`
- The `commandBuffer` parameter in `vkCmdEndRenderPass2KHR`
- The `swapchain` parameter in `vkGetSwapchainStatusKHR`
- The `yCbCrConversion` parameter in `vkDestroySamplerYcbcrConversionKHR`
- The `commandBuffer` parameter in `vkCmdDrawIndirectCountKHR`
- The `commandBuffer` parameter in `vkCmdDrawIndexedIndirectCountKHR`
- The `callback` parameter in `vkDestroyDebugReportCallbackEXT`
- The `object` member of the `pTagInfo` parameter in `vkDebugMarkerSetObjectTagEXT`
- The `object` member of the `pNameInfo` parameter in `vkDebugMarkerSetObjectNameEXT`
- The `commandBuffer` parameter in `vkCmdBindTransformFeedbackBuffersEXT`
- The `commandBuffer` parameter in `vkCmdBeginTransformFeedbackEXT`
- The `commandBuffer` parameter in `vkCmdEndTransformFeedbackEXT`
- The `commandBuffer` parameter in `vkCmdBeginQueryIndexedEXT`
- The `commandBuffer` parameter in `vkCmdEndQueryIndexedEXT`
- The `commandBuffer` parameter in `vkCmdDrawIndirectByteCountEXT`
- The `commandBuffer` parameter in `vkCmdDrawIndirectCountAMD`
- The `commandBuffer` parameter in `vkCmdDrawIndexedIndirectCountAMD`
- The `commandBuffer` parameter in `vkCmdBeginConditionalRenderingEXT`
- The `commandBuffer` parameter in `vkCmdEndConditionalRenderingEXT`
- The `commandBuffer` parameter in `vkCmdProcessCommandsNVX`
- The `commandBuffer` parameter in `vkCmdReserveSpaceForCommandsNVX`
- The `objectTable` parameter in `vkDestroyObjectTableNVX`
- The `objectTable` parameter in `vkRegisterObjectsNVX`
- The `objectTable` parameter in `vkUnregisterObjectsNVX`
- The `commandBuffer` parameter in `vkCmdSetViewportWScalingNV`
- The `swapchain` parameter in `vkGetRefreshCycleDurationGOOGLE`

- The `swapchain` parameter in `vkGetPastPresentationTimingGOOGLE`
- The `commandBuffer` parameter in `vkCmdSetDiscardRectangleEXT`
- The `objectHandle` member of the `pNameInfo` parameter in `vkSetDebugUtilsObjectNameEXT`
- The `objectHandle` member of the `pTagInfo` parameter in `vkSetDebugUtilsObjectTagEXT`
- The `messenger` parameter in `vkDestroyDebugUtilsMessengerEXT`
- The `commandBuffer` parameter in `vkCmdSetSampleLocationsEXT`
- The `validationCache` parameter in `vkDestroyValidationCacheEXT`
- The `dstCache` parameter in `vkMergeValidationCachesEXT`
- The `commandBuffer` parameter in `vkCmdBindShadingRateImageNV`
- The `commandBuffer` parameter in `vkCmdSetViewportShadingRatePaletteNV`
- The `commandBuffer` parameter in `vkCmdSetCoarseSampleOrderNV`
- The `commandBuffer` parameter in `vkCmdWriteBufferMarkerAMD`
- The `commandBuffer` parameter in `vkCmdDrawMeshTasksNV`
- The `commandBuffer` parameter in `vkCmdDrawMeshTasksIndirectNV`
- The `commandBuffer` parameter in `vkCmdDrawMeshTasksIndirectCountNV`
- The `commandBuffer` parameter in `vkCmdSetExclusiveScissorNV`
- The `commandBuffer` parameter in `vkCmdSetLineStippleEXT`

There are also a few instances where a command **can** take in a user allocated list whose contents are externally synchronized parameters. In these cases, the caller **must** guarantee that at most one thread is using a given element within the list at a given time. These parameters are listed below.

## Externally Synchronized Parameter Lists

- The `buffer` member of each element of the `pBufferBinds` member of each element of the `pBindInfo` parameter in [vkQueueBindSparse](#)
- The `image` member of each element of the `pImageOpaqueBinds` member of each element of the `pBindInfo` parameter in [vkQueueBindSparse](#)
- The `image` member of each element of the `pImageBinds` member of each element of the `pBindInfo` parameter in [vkQueueBindSparse](#)
- Each element of the `pFences` parameter in [vkResetFences](#)
- Each element of the `pDescriptorSets` parameter in [vkFreeDescriptorSets](#)
- The `dstSet` member of each element of the `pDescriptorWrites` parameter in [vkUpdateDescriptorSets](#)
- The `dstSet` member of each element of the `pDescriptorCopies` parameter in [vkUpdateDescriptorSets](#)
- Each element of the `pCommandBuffers` parameter in [vkFreeCommandBuffers](#)
- Each element of the `pWaitSemaphores` member of the `pPresentInfo` parameter in [vkQueuePresentKHR](#)
- Each element of the `pSwapchains` member of the `pPresentInfo` parameter in [vkQueuePresentKHR](#)
- The `surface` member of each element of the `pCreateInfos` parameter in [vkCreateSharedSwapchainsKHR](#)
- The `oldSwapchain` member of each element of the `pCreateInfos` parameter in [vkCreateSharedSwapchainsKHR](#)

In addition, there are some implicit parameters that need to be externally synchronized. For example, all `commandBuffer` parameters that need to be externally synchronized imply that the `commandPool` that was passed in when creating that command buffer also needs to be externally synchronized. The implicit parameters and their associated object are listed below.

## Implicit Externally Synchronized Parameters

- All `VkQueue` objects created from `device` in `vkDeviceWaitIdle`
- Any `VkDescriptorSet` objects allocated from `descriptorPool` in `vkResetDescriptorPool`
- The `VkCommandPool` that `commandBuffer` was allocated from in `vkBeginCommandBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from in `vkEndCommandBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindPipeline`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetViewport`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetScissor`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetLineWidth`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDepthBias`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetBlendConstants`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDepthBounds`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetStencilCompareMask`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetStencilWriteMask`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetStencilReference`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindDescriptorSets`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindIndexBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindVertexBuffers`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDraw`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexed`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirect`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexedIndirect`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDispatch`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDispatchIndirect`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyImage`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBlitImage`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyBufferToImage`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyImageToBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdUpdateBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdFillBuffer`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdClearColorImage`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdClearDepthImage`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdClearIndexImage`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdClearStencilImage`

## [vkCmdClearDepthStencilImage](#)

- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdClearAttachments`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdResolveImage`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetEvent`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdResetEvent`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdWaitEvents`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdPipelineBarrier`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginQuery`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndQuery`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdResetQueryPool`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdWriteTimestamp`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyQueryPoolResults`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdPushConstants`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginRenderPass`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdNextSubpass`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndRenderPass`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdExecuteCommands`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDeviceMask`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDispatchBase`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDeviceMaskKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDispatchBaseKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdPushDescriptorSetKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdPushDescriptorSetWithTemplateKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginRenderPass2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdNextSubpass2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndRenderPass2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirectCountKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexedIndirectCountKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDebugMarkerBeginEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDebugMarkerEndEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDebugMarkerInsertEXT`

- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindTransformFeedbackBuffersEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginTransformFeedbackEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndTransformFeedbackEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginQueryIndexedEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndQueryIndexedEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirectByteCountEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirectCountAMD`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexedIndirectCountAMD`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginConditionalRenderingEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndConditionalRenderingEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdProcessCommandsNVX`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdReserveSpaceForCommandsNVX`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetViewportWScalingNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDiscardRectangleEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginDebugUtilsLabelEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndDebugUtilsLabelEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdInsertDebugUtilsLabelEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetSampleLocationsEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindShadingRateImageNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetViewportShadingRatePaletteNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetCoarseSampleOrderNV`

- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBuildAccelerationStructureNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyAccelerationStructureNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdTraceRaysNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdWriteAccelerationStructuresPropertiesNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdWriteBufferMarkerAMD`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawMeshTasksNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawMeshTasksIndirectNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawMeshTasksIndirectCountNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetExclusiveScissorNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetCheckpointNV`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetPerformanceMarkerINTEL`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetPerformanceStreamMarkerINTEL`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetPerformanceOverrideINTEL`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetLineStippleEXT`

## 2.7. Errors

Vulkan is a layered API. The lowest layer is the core Vulkan layer, as defined by this Specification. The application **can** use additional layers above the core for debugging, validation, and other purposes.

One of the core principles of Vulkan is that building and submitting command buffers **should** be highly efficient. Thus error checking and validation of state in the core layer is minimal, although more rigorous validation **can** be enabled through the use of layers.

The core layer assumes applications are using the API correctly. Except as documented elsewhere in the Specification, the behavior of the core layer to an application using the API incorrectly is undefined, and **may** include program termination. However, implementations **must** ensure that incorrect usage by an application does not affect the integrity of the operating system, the Vulkan implementation, or other Vulkan client applications in the system. In particular, any guarantees made by an operating system about whether memory from one process **can** be visible to another process or not **must** not be violated by a Vulkan implementation for **any memory allocation**. Vulkan implementations are not **required** to make additional security or integrity guarantees

beyond those provided by the OS unless explicitly directed by the application's use of a particular feature or extension.

*Note*

For instance, if an operating system guarantees that data in all its memory allocations are set to zero when newly allocated, the Vulkan implementation **must** make the same guarantees for any allocations it controls (e.g. `VkDeviceMemory`).



Similarly, if an operating system guarantees that use-after-free of host allocations will not result in values written by another process becoming visible, the same guarantees **must** be made by the Vulkan implementation for device memory.

If the [protected memory](#) feature is supported, the implementation provides additional guarantees when invalid usage occurs to prevent values in protected memory from being accessed or inferred outside of protected operations, as described in [Protected Memory Access Rules](#).

Validation of correct API usage is left to validation layers. Applications **should** be developed with validation layers enabled, to help catch and eliminate errors. Once validated, released applications **should** not enable validation layers by default.

### 2.7.1. Valid Usage

Valid usage defines a set of conditions which **must** be met in order to achieve well-defined run-time behavior in an application. These conditions depend only on Vulkan state, and the parameters or objects whose usage is constrained by the condition.

Some valid usage conditions have dependencies on run-time limits or feature availability. It is possible to validate these conditions against Vulkan's minimum supported values for these limits and features, or some subset of other known values.

Valid usage conditions do not cover conditions where well-defined behavior (including returning an error code) exists.

Valid usage conditions **should** apply to the command or structure where complete information about the condition would be known during execution of an application. This is such that a validation layer or linter **can** be written directly against these statements at the point they are specified.

#### Note

This does lead to some non-obvious places for valid usage statements. For instance, the valid values for a structure might depend on a separate value in the calling command. In this case, the structure itself will not reference this valid usage as it is impossible to determine validity from the structure that it is invalid - instead this valid usage would be attached to the calling command.



Another example is draw state - the state setters are independent, and can cause a legitimately invalid state configuration between draw calls; so the valid usage statements are attached to the place where all state needs to be valid - at the draw command.

Valid usage conditions are described in a block labelled “Valid Usage” following each command or structure they apply to.

### 2.7.2. Implicit Valid Usage

Some valid usage conditions apply to all commands and structures in the API, unless explicitly denoted otherwise for a specific command or structure. These conditions are considered *implicit*, and are described in a block labelled “Valid Usage (Implicit)” following each command or structure they apply to. Implicit valid usage conditions are described in detail below.

#### Valid Usage for Object Handles

Any input parameter to a command that is an object handle **must** be a valid object handle, unless otherwise specified. An object handle is valid if:

- It has been created or allocated by a previous, successful call to the API. Such calls are noted in the Specification.
- It has not been deleted or freed by a previous call to the API. Such calls are noted in the Specification.
- Any objects used by that object, either as part of creation or execution, **must** also be valid.

The reserved values `VK_NULL_HANDLE` and `NULL` **can** be used in place of valid non-dispatchable handles and dispatchable handles, respectively, when *explicitly called out in the Specification*. Any command that creates an object successfully **must** not return these values. It is valid to pass these values to `vkDestroy*` or `vkFree*` commands, which will silently ignore these values.

#### Valid Usage for Pointers

Any parameter that is a pointer **must** be a *valid pointer* only if it is explicitly called out by a Valid Usage statement.

A pointer is “valid” if it points at memory containing values of the number and type(s) expected by the command, and all fundamental types accessed through the pointer (e.g. as elements of an array or as members of a structure) satisfy the alignment requirements of the host processor.

## Valid Usage for Strings

Any parameter that is a pointer to `char` **must** be a finite sequence of values terminated by a null character, or if *explicitly called out in the Specification*, **can** be `NULL`.

## Valid Usage for Enumerated Types

Any parameter of an enumerated type **must** be a valid enumerant for that type. A enumerant is valid if:

- The enumerant is defined as part of the enumerated type.
- The enumerant is not one of the special values defined for the enumerated type, which are suffixed with `_BEGIN_RANGE`, `_END_RANGE`, `_RANGE_SIZE` or `_MAX_ENUM`<sup>1</sup>.

1

The meaning of these special tokens is not exposed in the Vulkan Specification. They are not part of the API, and they **should** not be used by applications. Their original intended use was for internal consumption by Vulkan implementations. Even that use will no longer be supported in the future, but they will be retained for backwards compatibility reasons.

Any enumerated type returned from a query command or otherwise output from Vulkan to the application **must** not have a reserved value. Reserved values are values not defined by any extension for that enumerated type.

*Note*



This language is intended to accommodate cases such as “hidden” extensions known only to driver internals, or layers enabling extensions without knowledge of the application, without allowing return of values not defined by any extension.

## Valid Usage for Flags

A collection of flags is represented by a bitmask using the type `VkFlags`:

```
typedef uint32_t VkFlags;
```

Bitmasks are passed to many commands and structures to compactly represent options, but `VkFlags` is not used directly in the API. Instead, a `Vk*Flags` type which is an alias of `VkFlags`, and whose name matches the corresponding `Vk*FlagBits` that are valid for that type, is used.

Any `Vk*Flags` member or parameter used in the API as an input **must** be a valid combination of bit flags. A valid combination is either zero or the bitwise OR of valid bit flags. A bit flag is valid if:

- The bit flag is defined as part of the `Vk*FlagBits` type, where the bits type is obtained by taking the flag type and replacing the trailing `Flags` with `FlagBits`. For example, a flag value of type `VkColorComponentFlags` **must** contain only bit flags defined by `VkColorComponentFlagBits`.
- The flag is allowed in the context in which it is being used. For example, in some cases, certain bit flags or combinations of bit flags are mutually exclusive.

Any `Vk*Flags` member or parameter returned from a query command or otherwise output from Vulkan to the application **may** contain bit flags undefined in its corresponding `Vk*FlagBits` type. An application **cannot** rely on the state of these unspecified bits.

## Valid Usage for Structure Types

Any parameter that is a structure containing a `sType` member **must** have a value of `sType` which is a valid `VkStructureType` value matching the type of the structure.

Structure types supported by the Vulkan API include:

```
typedef enum VkStructureType {
    VK_STRUCTURE_TYPE_APPLICATION_INFO = 0,
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO = 1,
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO = 2,
    VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO = 3,
    VK_STRUCTURE_TYPE_SUBMIT_INFO = 4,
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO = 5,
    VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE = 6,
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO = 7,
    VK_STRUCTURE_TYPE_FENCE_CREATE_INFO = 8,
    VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO = 9,
    VK_STRUCTURE_TYPE_EVENT_CREATE_INFO = 10,
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO = 11,
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO = 12,
    VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO = 13,
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO = 14,
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO = 15,
    VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO = 16,
    VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO = 17,
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO = 18,
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO = 19,
    VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO = 20,
    VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO = 21,
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO = 22,
    VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO = 23,
    VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO = 24,
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO = 25,
    VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO = 26,
    VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO = 27,
    VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO = 28,
    VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO = 29,
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO = 30,
    VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO = 31,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO = 32,
    VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO = 33,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO = 34,
    VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET = 35,
    VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET = 36,
    VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO = 37,
```

```
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO = 38,
VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO = 39,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO = 40,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO = 41,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO = 42,
VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO = 43,
VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER = 44,
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER = 45,
VK_STRUCTURE_TYPE_MEMORY_BARRIER = 46,
VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO = 47,
VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO = 48,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES = 1000094000,
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO = 1000157000,
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO = 1000157001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES = 1000083000,
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS = 1000127000,
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO = 1000127001,
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO = 1000060000,
VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO = 1000060003,
VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO = 1000060004,
VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO = 1000060005,
VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO = 1000060006,
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO = 1000060013,
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO = 1000060014,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES = 1000070000,
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO = 1000070001,
VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2 = 1000146000,
VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2 = 1000146001,
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2 = 1000146002,
VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2 = 1000146003,
VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2 = 1000146004,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2 = 1000059000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2 = 1000059001,
VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2 = 1000059002,
VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2 = 1000059003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2 = 1000059004,
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2 = 1000059005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2 = 1000059006,
VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2 = 1000059007,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2 = 1000059008,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES = 1000117000,
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO = 1000117001,
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO = 1000117002,
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO =
1000117003,
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO = 1000053000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES = 1000053001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES = 1000053002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES = 1000120000,
VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO = 1000145000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES = 1000145001,
```

```
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES = 1000145002,
VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2 = 1000145003,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO = 1000156000,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO = 1000156001,
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO = 1000156002,
VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO = 1000156003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES = 1000156004,
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES = 1000156005,
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO = 1000085000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO = 1000071000,
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES = 1000071001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO = 1000071002,
VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES = 1000071003,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES = 1000071004,
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO = 1000072000,
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO = 1000072001,
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO = 1000072002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO = 1000112000,
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES = 1000112001,
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO = 1000113000,
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO = 1000077000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO = 1000076000,
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES = 1000076001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES = 1000168000,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT = 1000168001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES = 1000063000,
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR = 1000001000,
VK_STRUCTURE_TYPE_PRESENT_INFO_KHR = 1000001001,
VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR = 1000060007,
VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR = 1000060008,
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR = 1000060009,
VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR = 1000060010,
VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR = 1000060011,
VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR = 1000060012,
VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR = 1000002000,
VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR = 1000002001,
VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR = 1000003000,
VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR = 1000004000,
VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR = 1000005000,
VK_STRUCTURE_TYPE_WAYLAND_SURFACE_CREATE_INFO_KHR = 1000006000,
VK_STRUCTURE_TYPE_ANDROID_SURFACE_CREATE_INFO_KHR = 1000008000,
VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR = 1000009000,
VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT = 1000011000,
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_RASTERIZATION_ORDER_AMD =
1000018000,
VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_NAME_INFO_EXT = 1000022000,
VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_TAG_INFO_EXT = 1000022001,
VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT = 1000022002,
VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV = 1000026000,
VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_BUFFER_CREATE_INFO_NV = 1000026001,
VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV = 1000026002,
```

```

VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_FEATURES_EXT = 1000028000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_PROPERTIES_EXT = 1000028001,
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_STREAM_CREATE_INFO_EXT =
1000028002,
VK_STRUCTURE_TYPE_IMAGE_VIEW_HANDLE_INFO_NVX = 1000030000,
VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD = 1000041000,
VK_STRUCTURE_TYPE_STREAM_DESCRIPTOR_SURFACE_CREATE_INFO_GGP = 1000049000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CORNER_SAMPLED_IMAGE_FEATURES_NV =
1000050000,
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV = 1000056000,
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV = 1000056001,
VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_NV = 1000057000,
VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_NV = 1000057001,
VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_NV = 1000058000,
VK_STRUCTURE_TYPE_VALIDATION_FLAGS_EXT = 1000061000,
VK_STRUCTURE_TYPE_VI_SURFACE_CREATE_INFO_NN = 1000062000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT =
1000066000,
VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT = 1000067000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT = 1000067001,
VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHR = 1000073000,
VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR = 1000073001,
VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHR = 1000073002,
VK_STRUCTURE_TYPE_MEMORY_GET_WIN32_HANDLE_INFO_KHR = 1000073003,
VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHR = 1000074000,
VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR = 1000074001,
VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR = 1000074002,
VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHR = 1000075000,
VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR = 1000078000,
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR = 1000078001,
VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHR = 1000078002,
VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_KHR = 1000078003,
VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHR = 1000079000,
VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR = 1000079001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PUSH_DESCRIPTOR_PROPERTIES_KHR =
1000080000,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_CONDITIONAL_RENDERING_INFO_EXT =
1000081000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONDITIONAL_RENDERING_FEATURES_EXT =
1000081001,
VK_STRUCTURE_TYPE_CONDITIONAL_RENDERING_BEGIN_INFO_EXT = 1000081002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES_KHR =
1000082000,
VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR = 1000084000,
VK_STRUCTURE_TYPE_OBJECT_TABLE_CREATE_INFO_NVX = 1000086000,
VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_CREATE_INFO_NVX = 1000086001,
VK_STRUCTURE_TYPE_CMD_PROCESS_COMMANDS_INFO_NVX = 1000086002,
VK_STRUCTURE_TYPE_CMD_RESERVE_SPACE_FOR_COMMANDS_INFO_NVX = 1000086003,
VK_STRUCTURE_TYPE_DEVICE_GENERATED_COMMANDS_LIMITS_NVX = 1000086004,
VK_STRUCTURE_TYPE_DEVICE_GENERATED_COMMANDS_FEATURES_NVX = 1000086005,
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV =
1000087000,
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT = 1000090000,
VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT = 1000091000,
VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT = 1000091001,
VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT = 1000091002,

```

```
VK_STRUCTURE_TYPE_SWAPCHAIN_COUNTER_CREATE_INFO_EXT = 1000091003,
VK_STRUCTURE_TYPE_PRESENT_TIMES_INFO_GOOGLE = 1000092000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PER_VIEW_ATTRIBUTES_PROPERTIES_NVX =
1000097000,
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SWIZZLE_STATE_CREATE_INFO_NV = 1000098000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT = 1000099000,
VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT = 1000099001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_EXT =
1000101000,
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT =
1000101001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_CLIP_ENABLE_FEATURES_EXT = 1000102000,
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT =
1000102001,
VK_STRUCTURE_TYPE_HDR_METADATA_EXT = 1000105000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES_KHR = 1000108000,
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO_KHR = 1000108001,
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO_KHR = 1000108002,
VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO_KHR = 1000108003,
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2_KHR = 1000109000,
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2_KHR = 1000109001,
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2_KHR = 1000109002,
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2_KHR = 1000109003,
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2_KHR = 1000109004,
VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO_KHR = 1000109005,
VK_STRUCTURE_TYPE_SUBPASS_END_INFO_KHR = 1000109006,
VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR = 1000111000,
VK_STRUCTURE_TYPE_IMPORT_FENCE_WIN32_HANDLE_INFO_KHR = 1000114000,
VK_STRUCTURE_TYPE_EXPORT_FENCE_WIN32_HANDLE_INFO_KHR = 1000114001,
VK_STRUCTURE_TYPE_FENCE_GET_WIN32_HANDLE_INFO_KHR = 1000114002,
VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR = 1000115000,
VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR = 1000115001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR = 1000116000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR = 1000116001,
VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR = 1000116002,
VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR = 1000116003,
VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR = 1000116004,
VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR = 1000116005,
VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR = 1000116006,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR = 1000119000,
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR = 1000119001,
VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR = 1000119002,
VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR = 1000121000,
VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR = 1000121001,
VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR = 1000121002,
VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR = 1000121003,
VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR = 1000121004,
VK_STRUCTURE_TYPE_IOS_SURFACE_CREATE_INFO_MVK = 1000122000,
VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK = 1000123000,
VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT = 1000128000,
VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT = 1000128001,
```

```
VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT = 1000128002,
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT = 1000128003,
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT = 1000128004,
VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_USAGE_ANDROID = 1000129000,
VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_PROPERTIES_ANDROID = 1000129001,
VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_FORMAT_PROPERTIES_ANDROID = 1000129002,
VK_STRUCTURE_TYPE_IMPORT_ANDROID_HARDWARE_BUFFER_INFO_ANDROID = 1000129003,
VK_STRUCTURE_TYPE_MEMORY_GET_ANDROID_HARDWARE_BUFFER_INFO_ANDROID = 1000129004,
VK_STRUCTURE_TYPE_EXTERNAL_FORMAT_ANDROID = 1000129005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES_EXT =
1000130000,
VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO_EXT = 1000130001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES_EXT = 1000138000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES_EXT =
1000138001,
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT = 1000138002,
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO_EXT =
1000138003,
VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT = 1000143000,
VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT = 1000143001,
VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT = 1000143002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT = 1000143003,
VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT = 1000143004,
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO_KHR = 1000147000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_FEATURES_EXT =
1000148000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT =
1000148001,
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT =
1000148002,
VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_TO_COLOR_STATE_CREATE_INFO_NV = 1000149000,
VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_MODULATION_STATE_CREATE_INFO_NV = 1000152000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_FEATURES_NV = 1000154000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_PROPERTIES_NV = 1000154001,
VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT = 1000158000,
VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT = 1000158001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT = 1000158002,
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT = 1000158003,
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT = 1000158004,
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT = 1000158005,
VK_STRUCTURE_TYPE_VALIDATION_CACHE_CREATE_INFO_EXT = 1000160000,
VK_STRUCTURE_TYPE_SHADER_MODULE_VALIDATION_CACHE_CREATE_INFO_EXT = 1000160001,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO_EXT =
1000161000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES_EXT = 1000161001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES_EXT = 1000161002,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO_EXT =
1000161003,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT_EXT =
1000161004,
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SHADING_RATE_IMAGE_STATE_CREATE_INFO_NV =
```

```
1000164000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_FEATURES_NV = 1000164001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_PROPERTIES_NV = 1000164002,
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_COARSE_SAMPLE_ORDER_STATE_CREATE_INFO_NV =
1000164005,
VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_NV = 1000165000,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_NV = 1000165001,
VK_STRUCTURE_TYPE_GEOMETRY_NV = 1000165003,
VK_STRUCTURE_TYPE_GEOMETRY_TRIANGLES_NV = 1000165004,
VK_STRUCTURE_TYPE_GEOMETRY_AABB_NV = 1000165005,
VK_STRUCTURE_TYPE_BIND_ACCELERATION_STRUCTURE_MEMORY_INFO_NV = 1000165006,
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_NV = 1000165007,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_INFO_NV = 1000165008,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PROPERTIES_NV = 1000165009,
VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_NV = 1000165011,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_INFO_NV = 1000165012,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_REPRESENTATIVE_FRAGMENT_TEST_FEATURES_NV =
1000166000,
VK_STRUCTURE_TYPE_PIPELINE_REPRESENTATIVE_FRAGMENT_TEST_STATE_CREATE_INFO_NV =
1000166001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_VIEW_IMAGE_FORMAT_INFO_EXT = 1000170000,
VK_STRUCTURE_TYPE_FILTER_CUBIC_IMAGE_VIEW_IMAGE_FORMAT_PROPERTIES_EXT =
1000170001,
VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT = 1000174000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES_KHR =
1000175000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES_KHR = 1000177000,
VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT = 1000178000,
VK_STRUCTURE_TYPE_MEMORY_HOST_POINTER_PROPERTIES_EXT = 1000178001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT =
1000178002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES_KHR = 1000180000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CLOCK_FEATURES_KHR = 1000181000,
VK_STRUCTURE_TYPE_PIPELINE_COMPILER_CONTROL_CREATE_INFO_AMD = 1000183000,
VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT = 1000184000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_AMD = 1000185000,
VK_STRUCTURE_TYPE_DEVICE_MEMORY_OVERALLOCATION_CREATE_INFO_AMD = 1000189000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_PROPERTIES_EXT =
1000190000,
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT =
1000190001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_FEATURES_EXT =
1000190002,
VK_STRUCTURE_TYPE_PRESENT_FRAME_TOKEN_GGP = 1000191000,
VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO_EXT = 1000192000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES_KHR = 1000196000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES_KHR = 1000197000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES_KHR =
1000199000,
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE_KHR = 1000199001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COMPUTE_SHADER_DERIVATIVES_FEATURES_NV =
```

```

1000201000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_FEATURES_NV = 1000202000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_PROPERTIES_NV = 1000202001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_BARYCENTRIC_FEATURES_NV =
1000203000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_FOOTPRINT_FEATURES_NV = 1000204000,
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_EXCLUSIVE_SCISSOR_STATE_CREATE_INFO_NV =
1000205000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXCLUSIVE_SCISSOR_FEATURES_NV = 1000205002,
VK_STRUCTURE_TYPE_CHECKPOINT_DATA_NV = 1000206000,
VK_STRUCTURE_TYPE_QUEUE_FAMILY_CHECKPOINT_PROPERTIES_NV = 1000206001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES_KHR = 1000207000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES_KHR = 1000207001,
VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO_KHR = 1000207002,
VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO_KHR = 1000207003,
VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO_KHR = 1000207004,
VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO_KHR = 1000207005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_FUNCTIONS_2_FEATURES_INTEL =
1000209000,
VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO_INTEL = 1000210000,
VK_STRUCTURE_TYPE_INITIALIZE_PERFORMANCE_API_INFO_INTEL = 1000210001,
VK_STRUCTURE_TYPE_PERFORMANCE_MARKER_INFO_INTEL = 1000210002,
VK_STRUCTURE_TYPE_PERFORMANCE_STREAM_MARKER_INFO_INTEL = 1000210003,
VK_STRUCTURE_TYPE_PERFORMANCE_OVERRIDE_INFO_INTEL = 1000210004,
VK_STRUCTURE_TYPE_PERFORMANCE_CONFIGURATION_ACQUIRE_INFO_INTEL = 1000210005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES_KHR = 1000211000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PCI_BUS_INFO_PROPERTIES_EXT = 1000212000,
VK_STRUCTURE_TYPE_DISPLAY_NATIVE_HDR_SURFACE_CAPABILITIES_AMD = 1000213000,
VK_STRUCTURE_TYPE_SWAPCHAIN_DISPLAY_NATIVE_HDR_CREATE_INFO_AMD = 1000213001,
VK_STRUCTURE_TYPE_IMAGEPIPE_SURFACE_CREATE_INFO_FUCHSIA = 1000214000,
VK_STRUCTURE_TYPE_METAL_SURFACE_CREATE_INFO_EXT = 1000217000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_FEATURES_EXT = 1000218000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_PROPERTIES_EXT =
1000218001,
VK_STRUCTURE_TYPE_RENDER_PASS_FRAGMENT_DENSITY_MAP_CREATE_INFO_EXT = 1000218002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES_EXT = 1000221000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT =
1000225000,
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO_EXT =
1000225001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES_EXT = 1000225002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_2_AMD = 1000227000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COHERENT_MEMORY_FEATURES_AMD = 1000229000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT = 1000237000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PRIORITY_FEATURES_EXT = 1000238000,
VK_STRUCTURE_TYPE_MEMORY_PRIORITY_ALLOCATE_INFO_EXT = 1000238001,
VK_STRUCTURE_TYPE_SURFACE_PROTECTED_CAPABILITIES_KHR = 1000239000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEDICATED_ALLOCATION_IMAGE_ALIASING_FEATURES_NV
= 1000240000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES_KHR =
1000241000,

```

```

VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT_KHR = 1000241001,
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT_KHR = 1000241002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT = 1000244000,
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_CREATE_INFO_EXT = 1000244002,
VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO_EXT = 1000246000,
VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT = 1000247000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_FEATURES_NV = 1000249000,
VK_STRUCTURE_TYPE_COOPERATIVE_MATRIX_PROPERTIES_NV = 1000249001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_PROPERTIES_NV = 1000249002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COVERAGE_REDUCTION_MODE_FEATURES_NV =
1000250000,
VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_REDUCTION_STATE_CREATE_INFO_NV = 1000250001,
VK_STRUCTURE_TYPE_FRAMEBUFFER_MIXED_SAMPLES_COMBINATION_NV = 1000250002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT =
1000251000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT = 1000252000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES_KHR =
1000253000,
VK_STRUCTURE_TYPE_SURFACE_FULL_SCREEN_EXCLUSIVE_INFO_EXT = 1000255000,
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_FULL_SCREEN_EXCLUSIVE_EXT = 1000255002,
VK_STRUCTURE_TYPE_SURFACE_FULL_SCREEN_EXCLUSIVE_WIN32_INFO_EXT = 1000255001,
VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT = 1000256000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_KHR = 1000257000,
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR = 1000244001,
VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO_KHR = 1000257002,
VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO_KHR = 1000257003,
VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO_KHR = 1000257004,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT = 1000259000,
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT = 1000259001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_PROPERTIES_EXT = 1000259002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES_EXT = 1000261000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT = 1000265000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_EXECUTABLE_PROPERTIES_FEATURES_KHR =
1000269000,
VK_STRUCTURE_TYPE_PIPELINE_INFO_KHR = 1000269001,
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_PROPERTIES_KHR = 1000269002,
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INFO_KHR = 1000269003,
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_STATISTIC_KHR = 1000269004,
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INTERNAL REPRESENTATION_KHR = 1000269005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT =
1000276000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT =
1000281000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT =
1000281001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETER_FEATURES =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES,
VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT =
VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT,

```

```
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2,
VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2_KHR = VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2,
VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2_KHR =
VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2,
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2_KHR =
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2,
VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2_KHR =
VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2,
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO_KHR =
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO,
VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO_KHR =
VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO,
VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO_KHR =
VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO,
VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO_KHR =
VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO,
VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO_KHR =
VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO,
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO,
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES,
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO,
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO,
VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES,
```

```
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO,
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO,
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO,
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES,
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT16_INT8_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES_KHR,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES,
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO,
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES2_EXT =
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO,
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES,
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES,
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO,
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO,
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES,
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS_KHR =
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS,
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO_KHR =
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO,
VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2_KHR =
VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2,
VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2_KHR =
VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2,
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2_KHR =
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2,
VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2_KHR =
VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2,
VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2_KHR =
```

```

VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2,
    VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO,
    VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO_KHR =
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO,
    VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO,
    VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO_KHR =
VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES,
    VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES,
    VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO,
    VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT_KHR =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_ADDRESS_FEATURES_EXT =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT,
    VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_EXT =
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR,
    VK_STRUCTURE_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkStructureType;

```

Each value corresponds to a particular structure with a `sType` member with a matching name. As a general rule, the name of each `VkStructureType` value is obtained by taking the name of the structure, stripping the leading `Vk`, prefixing each capital letter with `_`, converting the entire resulting string to upper case, and prefixing it with `VK_STRUCTURE_TYPE_`. For example, structures of type `VkImageCreateInfo` correspond to a `VkStructureType` of `VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO`, and thus its `sType` member **must** equal that when it is passed to the API.

The values `VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO` and `VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO` are reserved for internal use by the loader, and do not have corresponding Vulkan structures in this Specification.

## Valid Usage for Structure Pointer Chains

Any parameter that is a structure containing a `void* pNext` member **must** have a value of `pNext` that is either `NULL`, or is a pointer to a valid structure defined by an extension, containing `sType` and `pNext` members as described in the [Vulkan Documentation and Extensions](#) document in the section “Extension Interactions”. The set of structures connected by `pNext` pointers is referred to as a `pNext chain`. If that extension is supported by the implementation, then it **must** be enabled.

Each type of valid structure **must** not appear more than once in a `pNext` chain.

Any component of the implementation (the loader, any enabled layers, and drivers) **must** skip over,

without processing (other than reading the `sType` and `pNext` members) any structures in the chain with `sType` values not defined by extensions supported by that component.

Extension structures are not described in the base Vulkan Specification, but either in layered Specifications incorporating those extensions, or in separate vendor-provided documents.

As a convenience to implementations and layers needing to iterate through a structure pointer chain, the Vulkan API provides two *base structures*. These structures allow for some type safety, and can be used by Vulkan API functions that operate on generic inputs and outputs.

The `VkBaseInStructure` structure is defined as:

```
typedef struct VkBaseInStructure {
    VkStructureType           sType;
    const struct VkBaseInStructure* pNext;
} VkBaseInStructure;
```

- `sType` is the structure type of the structure being iterated through.
- `pNext` is `NULL` or a pointer to the next structure in a structure chain.

`VkBaseInStructure` can be used to facilitate iterating through a read-only structure pointer chain.

The `VkBaseOutStructure` structure is defined as:

```
typedef struct VkBaseOutStructure {
    VkStructureType           sType;
    struct VkBaseOutStructure* pNext;
} VkBaseOutStructure;
```

- `sType` is the structure type of the structure being iterated through.
- `pNext` is `NULL` or a pointer to the next structure in a structure chain.

`VkBaseOutStructure` can be used to facilitate iterating through a structure pointer chain that returns data back to the application.

## Valid Usage for Nested Structures

The above conditions also apply recursively to members of structures provided as input to a command, either as a direct argument to the command, or themselves a member of another structure.

Specifics on valid usage of each command are covered in their individual sections.

## Valid Usage for Extensions

Instance-level functionality or behavior added by an `instance extension` to the API **must** not be used unless that extension is supported by the instance as determined by `vkEnumerateInstanceExtensionProperties`, and that extension is enabled in `VkInstanceCreateInfo`.

Physical-device-level functionality or behavior added by an [instance extension](#) to the API **must** not be used unless that extension is supported by the instance as determined by [vkEnumerateInstanceExtensionProperties](#), and that extension is enabled in [VkInstanceCreateInfo](#).

Physical-device-level functionality or behavior added by a [device extension](#) to the API **must** not be used unless the conditions described in [Extending Physical Device Core Functionality](#) are met.

Device functionality or behavior added by a [device extension](#) to the API **must** not be used unless that extension is supported by the device as determined by [vkEnumerateDeviceExtensionProperties](#), and that extension is enabled in [VkDeviceCreateInfo](#).

### Valid Usage for Newer Core Versions

Instance-level functionality or behavior added by a [new core version](#) of the API **must** not be used unless it is supported by the instance as determined by [vkEnumerateInstanceVersion](#) and the specified version of [VkApplicationInfo::apiVersion](#).

Physical-device-level functionality or behavior added by a [new core version](#) of the API **must** not be used unless it is supported by the physical device as determined by [VkPhysicalDeviceProperties::apiVersion](#) and the specified version of [VkApplicationInfo::apiVersion](#).

Device-level functionality or behavior added by a [new core version](#) of the API **must** not be used unless it is supported by the device as determined by [VkPhysicalDeviceProperties::apiVersion](#) and the specified version of [VkApplicationInfo::apiVersion](#).

### 2.7.3. Return Codes

While the core Vulkan API is not designed to capture incorrect usage, some circumstances still require return codes. Commands in Vulkan return their status via return codes that are in one of two categories:

- Successful completion codes are returned when a command needs to communicate success or status information. All successful completion codes are non-negative values.
- Run time error codes are returned when a command needs to communicate a failure that could only be detected at run time. All run time error codes are negative values.

All return codes in Vulkan are reported via [VkResult](#) return values. The possible codes are:

```

typedef enum VkResult {
    VK_SUCCESS = 0,
    VK_NOT_READY = 1,
    VK_TIMEOUT = 2,
    VK_EVENT_SET = 3,
    VK_EVENT_RESET = 4,
    VK_INCOMPLETE = 5,
    VK_ERROR_OUT_OF_HOST_MEMORY = -1,
    VK_ERROR_OUT_OF_DEVICE_MEMORY = -2,
    VK_ERROR_INITIALIZATION_FAILED = -3,
    VK_ERROR_DEVICE_LOST = -4,
    VK_ERROR_MEMORY_MAP_FAILED = -5,
    VK_ERROR_LAYER_NOT_PRESENT = -6,
    VK_ERROR_EXTENSION_NOT_PRESENT = -7,
    VK_ERROR_FEATURE_NOT_PRESENT = -8,
    VK_ERROR_INCOMPATIBLE_DRIVER = -9,
    VK_ERROR_TOO_MANY_OBJECTS = -10,
    VK_ERROR_FORMAT_NOT_SUPPORTED = -11,
    VK_ERROR_FRAGMENTED_POOL = -12,
    VK_ERROR_OUT_OF_POOL_MEMORY = -1000069000,
    VK_ERROR_INVALID_EXTERNAL_HANDLE = -1000072003,
    VK_ERROR_SURFACE_LOST_KHR = -1000000000,
    VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
    VK_SUBOPTIMAL_KHR = 1000001003,
    VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
    VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
    VK_ERROR_VALIDATION_FAILED_EXT = -1000011001,
    VK_ERROR_INVALID_SHADER_NV = -1000012000,
    VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT = -1000158000,
    VK_ERROR_FRAGMENTATION_EXT = -1000161000,
    VK_ERROR_NOT_PERMITTED_EXT = -1000174001,
    VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT = -1000255000,
    VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR = -1000244000,
    VK_ERROR_OUT_OF_POOL_MEMORY_KHR = VK_ERROR_OUT_OF_POOL_MEMORY,
    VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR = VK_ERROR_INVALID_EXTERNAL_HANDLE,
    VK_ERROR_INVALID_DEVICE_ADDRESS_EXT = VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR,
    VK_RESULT_MAX_ENUM = 0x7FFFFFFF
} VkResult;

```

### *Success Codes*

- **VK\_SUCCESS** Command successfully completed
- **VK\_NOT\_READY** A fence or query has not yet completed
- **VK\_TIMEOUT** A wait operation has not completed in the specified time
- **VK\_EVENT\_SET** An event is signaled
- **VK\_EVENT\_RESET** An event is unsignaled
- **VK\_INCOMPLETE** A return array was too small for the result

- **VK\_SUBOPTIMAL\_KHR** A swapchain no longer matches the surface properties exactly, but **can** still be used to present to the surface successfully.

#### Error codes

- **VK\_ERROR\_OUT\_OF\_HOST\_MEMORY** A host memory allocation has failed.
- **VK\_ERROR\_OUT\_OF\_DEVICE\_MEMORY** A device memory allocation has failed.
- **VK\_ERROR\_INITIALIZATION\_FAILED** Initialization of an object could not be completed for implementation-specific reasons.
- **VK\_ERROR\_DEVICE\_LOST** The logical or physical device has been lost. See [Lost Device](#)
- **VK\_ERROR\_MEMORY\_MAP\_FAILED** Mapping of a memory object has failed.
- **VK\_ERROR\_LAYER\_NOT\_PRESENT** A requested layer is not present or could not be loaded.
- **VK\_ERROR\_EXTENSION\_NOT\_PRESENT** A requested extension is not supported.
- **VK\_ERROR\_FEATURE\_NOT\_PRESENT** A requested feature is not supported.
- **VK\_ERROR\_INCOMPATIBLE\_DRIVER** The requested version of Vulkan is not supported by the driver or is otherwise incompatible for implementation-specific reasons.
- **VK\_ERROR\_TOO\_MANY\_OBJECTS** Too many objects of the type have already been created.
- **VK\_ERROR\_FORMAT\_NOT\_SUPPORTED** A requested format is not supported on this device.
- **VK\_ERROR\_FRAGMENTED\_POOL** A pool allocation has failed due to fragmentation of the pool's memory. This **must** only be returned if no attempt to allocate host or device memory was made to accommodate the new allocation. This **should** be returned in preference to **VK\_ERROR\_OUT\_OF\_POOL\_MEMORY**, but only if the implementation is certain that the pool allocation failure was due to fragmentation.
- **VK\_ERROR\_SURFACE\_LOST\_KHR** A surface is no longer available.
- **VK\_ERROR\_NATIVE\_WINDOW\_IN\_USE\_KHR** The requested window is already in use by Vulkan or another API in a manner which prevents it from being used again.
- **VK\_ERROR\_OUT\_OF\_DATE\_KHR** A surface has changed in such a way that it is no longer compatible with the swapchain, and further presentation requests using the swapchain will fail. Applications **must** query the new surface properties and recreate their swapchain if they wish to continue presenting to the surface.
- **VK\_ERROR\_INCOMPATIBLE\_DISPLAY\_KHR** The display used by a swapchain does not use the same presentable image layout, or is incompatible in a way that prevents sharing an image.
- **VK\_ERROR\_INVALID\_SHADER\_NV** One or more shaders failed to compile or link. More details are reported back to the application via [VK\\_EXT\\_debug\\_report](#) if enabled.
- **VK\_ERROR\_OUT\_OF\_POOL\_MEMORY** A pool memory allocation has failed. This **must** only be returned if no attempt to allocate host or device memory was made to accommodate the new allocation. If the failure was definitely due to fragmentation of the pool, **VK\_ERROR\_FRAGMENTED\_POOL** **should** be returned instead.
- **VK\_ERROR\_INVALID\_EXTERNAL\_HANDLE** An external handle is not a valid handle of the specified type.
- **VK\_ERROR\_FRAGMENTATION\_EXT** A descriptor pool creation has failed due to fragmentation.
- **VK\_ERROR\_INVALID\_DEVICE\_ADDRESS\_EXT** A buffer creation failed because the requested address is

not available.

- **VK\_ERROR\_INVALID\_OPAQUE\_CAPTURE\_ADDRESS\_KHR** A buffer creation or memory allocation failed because the requested address is not available.
- **VK\_ERROR\_FULL\_SCREEN\_EXCLUSIVE\_MODE\_LOST\_EXT** An operation on a swapchain created with **VK\_FULL\_SCREEN\_EXCLUSIVE\_APPLICATION\_CONTROLLED\_EXT** failed as it did not have exclusive full-screen access. This **may** occur due to implementation-dependent reasons, outside of the application's control.

If a command returns a run time error, unless otherwise specified any output parameters will have undefined contents, except that if the output parameter is a structure with **sType** and **pNext** fields, those fields will be unmodified. Any structures chained from **pNext** will also have undefined contents, except that **sType** and **pNext** will be unmodified.

Out of memory errors do not damage any currently existing Vulkan objects. Objects that have already been successfully created **can** still be used by the application.

Performance-critical commands generally do not have return codes. If a run time error occurs in such commands, the implementation will defer reporting the error until a specified point. For commands that record into command buffers (**vkCmd\***) run time errors are reported by **vkEndCommandBuffer**.

## 2.8. Numeric Representation and Computation

Implementations normally perform computations in floating-point, and **must** meet the range and precision requirements defined under “Floating-Point Computation” below.

These requirements only apply to computations performed in Vulkan operations outside of shader execution, such as texture image specification and sampling, and per-fragment operations. Range and precision requirements during shader execution differ and are specified by the [Precision and Operation of SPIR-V Instructions](#) section.

In some cases, the representation and/or precision of operations is implicitly limited by the specified format of vertex or texel data consumed by Vulkan. Specific floating-point formats are described later in this section.

### 2.8.1. Floating-Point Computation

Most floating-point computation is performed in SPIR-V shader modules. The properties of computation within shaders are constrained as defined by the [Precision and Operation of SPIR-V Instructions](#) section.

Some floating-point computation is performed outside of shaders, such as viewport and depth range calculations. For these computations, we do not specify how floating-point numbers are to be represented, or the details of how operations on them are performed, but only place minimal requirements on representation and precision as described in the remainder of this section.

We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1

part in  $10^5$ . The maximum representable magnitude for all floating-point values **must** be at least  $2^{32}$ .

$x \times 0 = 0 \times x = 0$  for any non-infinite and non-NaN  $x$ .

$1 \times x = x \times 1 = x$ .

$x + 0 = 0 + x = x$ .

$0^0 = 1$ .

Occasionally, further requirements will be specified. Most single-precision floating-point formats meet these requirements.

The special values Inf and -Inf encode values with magnitudes too large to be represented; the special value NaN encodes “Not A Number” values resulting from undefined arithmetic operations such as  $0 / 0$ . Implementations **may** support Inf and NaN in their floating-point computations.

## 2.8.2. Floating-Point Format Conversions

When a value is converted to a defined floating-point representation, finite values falling between two representable finite values are rounded to one or the other. The rounding mode is not defined. Finite values whose magnitude is larger than that of any representable finite value may be rounded either to the closest representable finite value or to the appropriately signed infinity. For unsigned destination formats any negative values are converted to zero. Positive infinity is converted to positive infinity; negative infinity is converted to negative infinity in signed formats and to zero in unsigned formats; and any NaN is converted to a NaN.

## 2.8.3. 16-Bit Floating-Point Numbers

16-bit floating point numbers are defined in the “16-bit floating point numbers” section of the [Khronos Data Format Specification](#).

## 2.8.4. Unsigned 11-Bit Floating-Point Numbers

Unsigned 11-bit floating point numbers are defined in the “Unsigned 11-bit floating point numbers” section of the [Khronos Data Format Specification](#).

## 2.8.5. Unsigned 10-Bit Floating-Point Numbers

Unsigned 10-bit floating point numbers are defined in the “Unsigned 10-bit floating point numbers” section of the [Khronos Data Format Specification](#).

## 2.8.6. General Requirements

Any representable floating-point value in the appropriate format is legal as input to a Vulkan command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but **must** not lead to Vulkan interruption or termination. For example, providing a negative zero (where applicable) or a denormalized number to an Vulkan command **must** yield deterministic results, while providing a NaN or Inf yields

unspecified results.

Some calculations require division. In such cases (including implied divisions performed by vector normalization), division by zero produces an unspecified result but **must** not lead to Vulkan interruption or termination.

## 2.9. Fixed-Point Data Conversions

When generic vertex attributes and pixel color or depth *components* are represented as integers, they are often (but not always) considered to be *normalized*. Normalized integer values are treated specially when being converted to and from floating-point values, and are usually referred to as *normalized fixed-point*.

In the remainder of this section,  $b$  denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined by the API,  $b$  is the bit width of that type. When the integer comes from an [image](#) containing color or depth component texels,  $b$  is the number of bits allocated to that component in its [specified image format](#).

The signed and unsigned fixed-point representations are assumed to be  $b$ -bit binary two's-complement integers and binary unsigned integers, respectively.

### 2.9.1. Conversion from Normalized Fixed-Point to Floating-Point

Unsigned normalized fixed-point integers represent numbers in the range [0,1]. The conversion from an unsigned normalized fixed-point value  $c$  to the corresponding floating-point value  $f$  is defined as

$$f = \frac{c}{2^b - 1}$$

Signed normalized fixed-point integers represent numbers in the range [-1,1]. The conversion from a signed normalized fixed-point value  $c$  to the corresponding floating-point value  $f$  is performed using

$$f = \max\left(\frac{c}{2^{b-1} - 1}, -1.0\right)$$

Only the range  $[-2^{b-1} + 1, 2^{b-1} - 1]$  is used to represent signed fixed-point values in the range [-1,1]. For example, if  $b = 8$ , then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. Note that while zero is exactly expressible in this representation, one value (-128 in the example) is outside the representable range, and **must** be clamped before use. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point.

### 2.9.2. Conversion from Floating-Point to Normalized Fixed-Point

The conversion from a floating-point value  $f$  to the corresponding unsigned normalized fixed-point value  $c$  is defined by first clamping  $f$  to the range [0,1], then computing

$$c = \text{convertFloatToInt}(f \times (2^b - 1), b)$$

where `convertFloatToInt(r,b)` returns one of the two unsigned binary integer values with exactly

$b$  bits which are closest to the floating-point value  $r$ . Implementations **should** round to nearest. If  $r$  is equal to an integer, then that integer value **must** be returned. In particular, if  $f$  is equal to 0.0 or 1.0, then  $c$  **must** be assigned 0 or  $2^{b-1} - 1$ , respectively.

The conversion from a floating-point value  $f$  to the corresponding signed normalized fixed-point value  $c$  is performed by clamping  $f$  to the range [-1,1], then computing

$$c = \text{convertFloatToInt}(f \times (2^{b-1} - 1), b)$$

where `convertFloatToInt(r,b)` returns one of the two signed two's-complement binary integer values with exactly  $b$  bits which are closest to the floating-point value  $r$ . Implementations **should** round to nearest. If  $r$  is equal to an integer, then that integer value **must** be returned. In particular, if  $f$  is equal to -1.0, 0.0, or 1.0, then  $c$  **must** be assigned  $-(2^{b-1} - 1)$ , 0, or  $2^{b-1} - 1$ , respectively.

This equation is used everywhere that floating-point values are converted to signed normalized fixed-point.

## 2.10. Common Object Types

Some types of Vulkan objects are used in many different structures and command parameters, and are described here. These types include *offsets*, *extents*, and *rectangles*.

### 2.10.1. Offsets

Offsets are used to describe a pixel location within an image or framebuffer, as an (x,y) location for two-dimensional images, or an (x,y,z) location for three-dimensional images.

A two-dimensional offset is defined by the structure:

```
typedef struct VkOffset2D {
    int32_t    x;
    int32_t    y;
} VkOffset2D;
```

- $x$  is the x offset.
- $y$  is the y offset.

A three-dimensional offset is defined by the structure:

```
typedef struct VkOffset3D {
    int32_t    x;
    int32_t    y;
    int32_t    z;
} VkOffset3D;
```

- $x$  is the x offset.
- $y$  is the y offset.

- `z` is the z offset.

## 2.10.2. Extents

Extents are used to describe the size of a rectangular region of pixels within an image or framebuffer, as (width,height) for two-dimensional images, or as (width,height,depth) for three-dimensional images.

A two-dimensional extent is defined by the structure:

```
typedef struct VkExtent2D {
    uint32_t    width;
    uint32_t    height;
} VkExtent2D;
```

- `width` is the width of the extent.
- `height` is the height of the extent.

A three-dimensional extent is defined by the structure:

```
typedef struct VkExtent3D {
    uint32_t    width;
    uint32_t    height;
    uint32_t    depth;
} VkExtent3D;
```

- `width` is the width of the extent.
- `height` is the height of the extent.
- `depth` is the depth of the extent.

## 2.10.3. Rectangles

Rectangles are used to describe a specified rectangular region of pixels within an image or framebuffer. Rectangles include both an offset and an extent of the same dimensionality, as described above. Two-dimensional rectangles are defined by the structure

```
typedef struct VkRect2D {
    VkOffset2D    offset;
    VkExtent2D    extent;
} VkRect2D;
```

- `offset` is a `VkOffset2D` specifying the rectangle offset.
- `extent` is a `VkExtent2D` specifying the rectangle extent.

# Chapter 3. Initialization

Before using Vulkan, an application **must** initialize it by loading the Vulkan commands, and creating a `VkInstance` object.

## 3.1. Command Function Pointers

Vulkan commands are not necessarily exposed by static linking on a platform. Commands to query function pointers for Vulkan commands are described below.

### Note

When extensions are [promoted](#) or otherwise incorporated into another extension or Vulkan core version, command [aliases](#) may be included. Whilst the behavior of each command alias is identical, the behavior of retrieving each alias's function pointer is not. A function pointer for a given alias can only be retrieved if the extension or version that introduced that alias is supported and enabled, irrespective of whether any other alias is available.



Function pointers for all Vulkan commands **can** be obtained with the command:

```
PFN_vkVoidFunction vkGetInstanceProcAddr(  
    VkInstance instance,  
    const char* pName);
```

- `instance` is the instance that the function pointer will be compatible with, or `NULL` for commands not dependent on any instance.
- `pName` is the name of the command to obtain.

`vkGetInstanceProcAddr` itself is obtained in a platform- and loader- specific manner. Typically, the loader library will export this command as a function symbol, so applications **can** link against the loader library, or load it dynamically and look up the symbol using platform-specific APIs.

The table below defines the various use cases for `vkGetInstanceProcAddr` and expected return value (“fp” is “function pointer”) for each case.

The returned function pointer is of type `PFN_vkVoidFunction`, and must be cast to the type of the command being queried.

*Table 1. `vkGetInstanceProcAddr` behavior*

<code>instance</code>	<code>pName</code>	<b>return value</b>
<code>*<sup>1</sup></code>	<code>NULL</code>	<code>undefined</code>
invalid non- <code>NULL</code> instance	<code>*<sup>1</sup></code>	<code>undefined</code>
<code>NULL</code>	<code>vkEnumerateInstanceVersion</code>	<code>fp</code>

instance	pName	return value
NULL	<code>vkEnumerateInstanceExtensionProperties</code>	fp
NULL	<code>vkEnumerateInstanceLayerProperties</code>	fp
NULL	<code>vkCreateInstance</code>	fp
instance	core Vulkan command	$fp^2$
instance	enabled instance extension commands for <code>instance</code>	$fp^2$
instance	available device extension <sup>3</sup> commands for <code>instance</code>	$fp^2$
any other case, not covered above		NULL

1

"\*\*" means any representable value for the parameter (including valid values, invalid values, and `NULL`).

2

The returned function pointer **must** only be called with a dispatchable object (the first parameter) that is `instance` or a child of `instance`, e.g. `VkInstance`, `VkPhysicalDevice`, `VkDevice`, `VkQueue`, or `VkCommandBuffer`.

3

An "available device extension" is a device extension supported by any physical device enumerated by `instance`.

### Valid Usage (Implicit)

- If `instance` is not `NULL`, `instance` **must** be a valid `VkInstance` handle
- `pName` **must** be a null-terminated UTF-8 string

In order to support systems with multiple Vulkan implementations, the function pointers returned by `vkGetInstanceProcAddr` **may** point to dispatch code that calls a different real implementation for different `VkDevice` objects or their child objects. The overhead of the internal dispatch for `VkDevice` objects can be avoided by obtaining device-specific function pointers for any commands that use a device or device-child object as their dispatchable object. Such function pointers **can** be obtained with the command:

```
PFN_vkVoidFunction vkGetDeviceProcAddr(
    VkDevice                         device,
    const char*                      pName);
```

The table below defines the various use cases for `vkGetDeviceProcAddr` and expected return value for each case.

The returned function pointer is of type `PFN_vkVoidFunction`, and must be cast to the type of the command being queried. The function pointer **must** only be called with a dispatchable object (the first parameter) that is `device` or a child of `device`.

Table 2. `vkGetDeviceProcAddr` behavior

<code>device</code>	<code>pName</code>	return value
<code>NULL</code>	<code>*<sup>1</sup></code>	undefined
invalid device	<code>*<sup>1</sup></code>	undefined
<code>device</code>	<code>NULL</code>	undefined
<code>device</code>	core device-level Vulkan command	<code>fp<sup>2</sup></code>
<code>device</code>	enabled device extension device-level commands	<code>fp<sup>2</sup></code>
any other case, not covered above		<code>NULL</code>

1

"\*\*" means any representable value for the parameter (including valid values, invalid values, and `NULL`).

2

The returned function pointer **must** only be called with a dispatchable object (the first parameter) that is `device` or a child of `device` e.g. `VkDevice`, `VkQueue`, or `VkCommandBuffer`.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pName` **must** be a null-terminated UTF-8 string

The definition of `PFN_vkVoidFunction` is:

```
typedef void (VKAPI_PTR *PFN_vkVoidFunction)(void);
```

### 3.1.1. Extending Physical Device Core Functionality

New core physical-device-level functionality **can** be used when the physical-device version is greater than or equal to the version of Vulkan that added the new functionality. The Vulkan version supported by a physical device **can** be obtained by calling `vkGetPhysicalDeviceProperties`.

### 3.1.2. Extending Physical Device From Device Extensions

When the `VK_KHR_get_physical_device_properties2` extension is enabled, or when both the instance

and the physical-device versions are at least 1.1, physical-device-level functionality of a device extension **can** be used with a physical device if the corresponding extension is enumerated by `vkEnumerateDeviceExtensionProperties` for that physical device, even before a logical device has been created.

To obtain a function pointer for a physical-device-level command from a device extension, an application **can** use `vkGetInstanceProcAddr`. This function pointer **may** point to dispatch code, which calls a different real implementation for different `VkPhysicalDevice` objects. Applications **must** not use a `VkPhysicalDevice` in any command added by an extension or core version that is not supported by that physical device.

Device extensions **may** define structures that **can** be added to the `pNext` chain of physical-device-level commands.

## 3.2. Instances

There is no global state in Vulkan and all per-application state is stored in a `VkInstance` object. Creating a `VkInstance` object initializes the Vulkan library and allows the application to pass information about itself to the implementation.

Instances are represented by `VkInstance` handles:

```
VK_DEFINE_HANDLE(VkInstance)
```

To query the version of instance-level functionality supported by the implementation, call:

```
VkResult vkEnumerateInstanceVersion(  
    uint32_t* pApiVersion);
```

- `pApiVersion` is a pointer to a `uint32_t`, which is the version of Vulkan supported by instance-level functionality, encoded as described in [Version Numbers](#).

### Valid Usage (Implicit)

- `pApiVersion` **must** be a valid pointer to a `uint32_t` value

### Return Codes

#### Success

- `VK_SUCCESS`

To create an instance object, call:

```
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkInstance* pInstance);
```

- `pCreateInfo` is a pointer to a `VkInstanceCreateInfo` structure controlling creation of the instance.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pInstance` points a `VkInstance` handle in which the resulting instance is returned.

`vkCreateInstance` verifies that the requested layers exist. If not, `vkCreateInstance` will return `VK_ERROR_LAYER_NOT_PRESENT`. Next `vkCreateInstance` verifies that the requested extensions are supported (e.g. in the implementation or in any enabled instance layer) and if any requested extension is not supported, `vkCreateInstance` **must** return `VK_ERROR_EXTENSION_NOT_PRESENT`. After verifying and enabling the instance layers and extensions the `VkInstance` object is created and returned to the application. If a requested extension is only supported by a layer, both the layer and the extension need to be specified at `vkCreateInstance` time for the creation to succeed.

## Valid Usage

- All required extensions for each extension in the `VkInstanceCreateInfo` `::ppEnabledExtensionNames` list **must** also be present in that list.

## Valid Usage (Implicit)

- `pCreateInfo` **must** be a valid pointer to a valid `VkInstanceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pInstance` **must** be a valid pointer to a `VkInstance` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`
- `VK_ERROR_LAYER_NOT_PRESENT`
- `VK_ERROR_EXTENSION_NOT_PRESENT`
- `VK_ERROR_INCOMPATIBLE_DRIVER`

The `VkInstanceCreateInfo` structure is defined as:

```
typedef struct VkInstanceCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkInstanceCreateFlags     flags;
    const VkApplicationInfo* pApplicationInfo;
    uint32_t                  enabledLayerCount;
    const char* const*        ppEnabledLayerNames;
    uint32_t                  enabledExtensionCount;
    const char* const*        ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `pApplicationInfo` is `NULL` or a pointer to a `VkApplicationInfo` structure. If not `NULL`, this information helps implementations recognize behavior inherent to classes of applications. `VkApplicationInfo` is defined in detail below.
- `enabledLayerCount` is the number of global layers to enable.
- `ppEnabledLayerNames` is a pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings containing the names of layers to enable for the created instance. See the [Layers](#) section for further details.
- `enabledExtensionCount` is the number of global extensions to enable.
- `ppEnabledExtensionNames` is a pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings containing the names of extensions to enable.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDebugReportCallbackCreateInfoEXT`, `VkDebugUtilsMessengerCreateInfoEXT`, `VkValidationFeaturesEXT`, or `VkValidationFlagsEXT`
- Each `sType` member in the `pNext` chain must be unique
- `flags` must be `0`
- If `pApplicationInfo` is not `NULL`, `pApplicationInfo` must be a valid pointer to a valid `VkApplicationInfo` structure
- If `enabledLayerCount` is not `0`, `ppEnabledLayerNames` must be a valid pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings
- If `enabledExtensionCount` is not `0`, `ppEnabledExtensionNames` must be a valid pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings

```
typedef VkFlags VkInstanceCreateFlags;
```

`VkInstanceCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

When creating a Vulkan instance for which you wish to disable validation checks, add a `VkValidationFlagsEXT` structure to the `pNext` chain of the `VkInstanceCreateInfo` structure, specifying the checks to be disabled.

```
typedef struct VkValidationFlagsEXT {
    VkStructureType          sType;
    const void*              pNext;
    uint32_t                 disabledValidationCheckCount;
    const VkValidationCheckEXT* pDisabledValidationChecks;
} VkValidationFlagsEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `disabledValidationCheckCount` is the number of checks to disable.
- `pDisabledValidationChecks` is a pointer to an array of `VkValidationCheckEXT` values specifying the validation checks to be disabled.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_VALIDATION_FLAGS_EXT`
- `pDisabledValidationChecks` must be a valid pointer to an array of `disabledValidationCheckCount` valid `VkValidationCheckEXT` values
- `disabledValidationCheckCount` must be greater than `0`

Possible values of elements of the `VkValidationFlagsEXT::pDisabledValidationChecks` array, specifying validation checks to be disabled, are:

```
typedef enum VkValidationCheckEXT {  
    VK_VALIDATION_CHECK_ALL_EXT = 0,  
    VK_VALIDATION_CHECK_SHADERS_EXT = 1,  
    VK_VALIDATION_CHECK_MAX_ENUM_EXT = 0x7FFFFFFF  
} VkValidationCheckEXT;
```

- `VK_VALIDATION_CHECK_ALL_EXT` specifies that all validation checks are disabled.
- `VK_VALIDATION_CHECK_SHADERS_EXT` specifies that shader validation is disabled.

When creating a Vulkan instance for which you wish to enable or disable specific validation features, add a `VkValidationFeaturesEXT` structure to the `pNext` chain of the `VkInstanceCreateInfo` structure, specifying the features to be enabled or disabled.

```
typedef struct VkValidationFeaturesEXT {  
    VkStructureType sType;  
    const void* pNext;  
    uint32_t enabledValidationFeatureCount;  
    const VkValidationFeatureEnableEXT* pEnabledValidationFeatures;  
    uint32_t disabledValidationFeatureCount;  
    const VkValidationFeatureDisableEXT* pDisabledValidationFeatures;  
} VkValidationFeaturesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `enabledValidationFeatureCount` is the number of features to enable.
- `pEnabledValidationFeatures` is a pointer to an array of `VkValidationFeatureEnableEXT` values specifying the validation features to be enabled.
- `disabledValidationFeatureCount` is the number of features to disable.
- `pDisabledValidationFeatures` is a pointer to an array of `VkValidationFeatureDisableEXT` values specifying the validation features to be disabled.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT`
- If `enabledValidationFeatureCount` is not `0`, `pEnabledValidationFeatures` must be a valid pointer to an array of `enabledValidationFeatureCount` valid `VkValidationFeatureEnableEXT` values
- If `disabledValidationFeatureCount` is not `0`, `pDisabledValidationFeatures` must be a valid pointer to an array of `disabledValidationFeatureCount` valid `VkValidationFeatureDisableEXT` values

Possible values of elements of the `VkValidationFeaturesEXT::pEnabledValidationFeatures` array, specifying validation features to be enabled, are:

```
typedef enum VkValidationFeatureEnableEXT {
    VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT = 0,
    VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT = 1,
    VK_VALIDATION_FEATURE_ENABLE_BEST_PRACTICES_EXT = 2,
    VK_VALIDATION_FEATURE_ENABLE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkValidationFeatureEnableEXT;
```

- `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT` specifies that GPU-assisted validation is enabled. Activating this feature instruments shader programs to generate additional diagnostic data. This feature is disabled by default.
- `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT` specifies that the validation layers reserve a descriptor set binding slot for their own use. The layer reports a value for `VkPhysicalDeviceLimits::maxBoundDescriptorSets` that is one less than the value reported by the device. If the device supports the binding of only one descriptor set, the validation layer does not perform GPU-assisted validation. This feature is disabled by default. The GPU-assisted validation feature must be enabled in order to use this feature.
- `VK_VALIDATION_FEATURE_ENABLE_BEST_PRACTICES_EXT` specifies that Vulkan best-practices validation is enabled. Activating this feature enables the output of warnings related to common misuse of the API, but which are not explicitly prohibited by the specification. This feature is disabled by default.

Possible values of elements of the `VkValidationFeaturesEXT::pDisabledValidationFeatures` array, specifying validation features to be disabled, are:

```

typedef enum VkValidationFeatureDisableEXT {
    VK_VALIDATION_FEATURE_DISABLE_ALL_EXT = 0,
    VK_VALIDATION_FEATURE_DISABLE_SHADERS_EXT = 1,
    VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT = 2,
    VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT = 3,
    VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT = 4,
    VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT = 5,
    VK_VALIDATION_FEATURE_DISABLE_UNIQUE_HANDLES_EXT = 6,
    VK_VALIDATION_FEATURE_DISABLE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkValidationFeatureDisableEXT;

```

- `VK_VALIDATION_FEATURE_DISABLE_ALL_EXT` specifies that all validation checks are disabled.
- `VK_VALIDATION_FEATURE_DISABLE_SHADERS_EXT` specifies that shader validation is disabled. This feature is enabled by default.
- `VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT` specifies that thread safety validation is disabled. This feature is enabled by default.
- `VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT` specifies that stateless parameter validation is disabled. This feature is enabled by default.
- `VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT` specifies that object lifetime validation is disabled. This feature is enabled by default.
- `VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT` specifies that core validation checks are disabled. This feature is enabled by default. If this feature is disabled, the shader validation and GPU-assisted validation features are also disabled.
- `VK_VALIDATION_FEATURE_DISABLE_UNIQUE_HANDLES_EXT` specifies that protection against duplicate non-dispatchable object handles is disabled. This feature is enabled by default.

*Note*



Disabling checks such as parameter validation and object lifetime validation prevents the reporting of error conditions that can cause other validation checks to behave incorrectly or crash. Some validation checks assume that their inputs are already valid and do not always revalidate them.

*Note*



The `VK_EXT_validation_features` extension subsumes all the functionality provided in the `VK_EXT_validation_flags` extension.

The `VkApplicationInfo` structure is defined as:

```
typedef struct VkApplicationInfo {
    VkStructureType sType;
    const void* pNext;
    const char* pApplicationName;
    uint32_t applicationVersion;
    const char* pEngineName;
    uint32_t engineVersion;
    uint32_t apiVersion;
} VkApplicationInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pApplicationName` is `NULL` or is a pointer to a null-terminated UTF-8 string containing the name of the application.
- `applicationVersion` is an unsigned integer variable containing the developer-supplied version number of the application.
- `pEngineName` is `NULL` or is a pointer to a null-terminated UTF-8 string containing the name of the engine (if any) used to create the application.
- `engineVersion` is an unsigned integer variable containing the developer-supplied version number of the engine used to create the application.
- `apiVersion` **must** be the highest version of Vulkan that the application is designed to use, encoded as described in [Version Numbers](#). The patch version number specified in `apiVersion` is ignored when creating an instance object. Only the major and minor versions of the instance **must** match those requested in `apiVersion`.

Vulkan 1.0 implementations were required to return `VK_ERROR_INCOMPATIBLE_DRIVER` if `apiVersion` was larger than 1.0. Implementations that support Vulkan 1.1 or later **must** not return `VK_ERROR_INCOMPATIBLE_DRIVER` for any value of `apiVersion`.

*Note*



Because Vulkan 1.0 implementations **may** fail with `VK_ERROR_INCOMPATIBLE_DRIVER`, applications **should** determine the version of Vulkan available before calling `vkCreateInstance`. If the `vkGetInstanceProcAddr` returns `NULL` for `vkEnumerateInstanceVersion`, it is a Vulkan 1.0 implementation. Otherwise, the application **can** call `vkEnumerateInstanceVersion` to determine the version of Vulkan.

As long as the instance supports at least Vulkan 1.1, an application **can** use different versions of Vulkan with an instance than it does with a device or physical device.

*Note*

The Khronos validation layers will treat `apiVersion` as the highest API version the application targets, and will validate API usage against the minimum of that version and the implementation version (instance or device, depending on context). If an application tries to use functionality from a greater version than this, a validation error will be triggered.

For example, if the instance supports Vulkan 1.1 and three physical devices support Vulkan 1.0, Vulkan 1.1, and a hypothetical Vulkan 1.2, respectively, and if the application sets `apiVersion` to 1.2, the application **can** use the following versions of Vulkan:

- Vulkan 1.0 **can** be used with the instance and with all physical devices.
- Vulkan 1.1 **can** be used with the instance and with the physical devices that support Vulkan 1.1 and Vulkan 1.2.
- Vulkan 1.2 **can** be used with the physical device that supports Vulkan 1.2.

If we modify the above example so that the application sets `apiVersion` to 1.1, then the application **must** not use Vulkan 1.2 functionality on the physical device that supports Vulkan 1.2.

Implicit layers **must** be disabled if they do not support a version at least as high as `apiVersion`. See the [Vulkan Loader Specification and Architecture Overview](#) document for additional information.

*Note*

Providing a `NULL` `VkInstanceCreateInfo::pApplicationInfo` or providing an `apiVersion` of 0 is equivalent to providing an `apiVersion` of `VK_MAKE_VERSION(1,0,0)`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_APPLICATION_INFO`
- `pNext` **must** be `NULL`
- If `pApplicationName` is not `NULL`, `pApplicationName` **must** be a null-terminated UTF-8 string
- If `pEngineName` is not `NULL`, `pEngineName` **must** be a null-terminated UTF-8 string

To destroy an instance, call:

```
void vkDestroyInstance(  
    VkInstance  
    const VkAllocationCallbacks*  
                                instance,  
                                pAllocator);
```

- `instance` is the handle of the instance to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All child objects created using `instance` **must** have been destroyed prior to destroying `instance`
- If `VkAllocationCallbacks` were provided when `instance` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `instance` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- If `instance` is not `NULL`, `instance` **must** be a valid `VkInstance` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure

## Host Synchronization

- Host access to `instance` **must** be externally synchronized

# Chapter 4. Devices and Queues

Once Vulkan is initialized, devices and queues are the primary objects used to interact with a Vulkan implementation.

Vulkan separates the concept of *physical* and *logical* devices. A physical device usually represents a single complete implementation of Vulkan (excluding instance-level functionality) available to the host, of which there are a finite number. A logical device represents an instance of that implementation with its own state and resources independent of other logical devices.

Physical devices are represented by `VkPhysicalDevice` handles:

```
VK_DEFINE_HANDLE(VkPhysicalDevice)
```

## 4.1. Physical Devices

To retrieve a list of physical device objects representing the physical devices installed in the system, call:

```
VkResult vkEnumeratePhysicalDevices(  
    VkInstance  
    uint32_t*  
    VkPhysicalDevice*  
        instance,  
        pPhysicalDeviceCount,  
        pPhysicalDevices);
```

- `instance` is a handle to a Vulkan instance previously created with `vkCreateInstance`.
- `pPhysicalDeviceCount` is a pointer to an integer related to the number of physical devices available or queried, as described below.
- `pPhysicalDevices` is either `NULL` or a pointer to an array of `VkPhysicalDevice` handles.

If `pPhysicalDevices` is `NULL`, then the number of physical devices available is returned in `pPhysicalDeviceCount`. Otherwise, `pPhysicalDeviceCount` **must** point to a variable set by the user to the number of elements in the `pPhysicalDevices` array, and on return the variable is overwritten with the number of handles actually written to `pPhysicalDevices`. If `pPhysicalDeviceCount` is less than the number of physical devices available, at most `pPhysicalDeviceCount` structures will be written. If `pPhysicalDeviceCount` is smaller than the number of physical devices available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available physical devices were returned.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pPhysicalDeviceCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPhysicalDeviceCount` is not `0`, and `pPhysicalDevices` is not `NULL`, `pPhysicalDevices` **must** be a valid pointer to an array of `pPhysicalDeviceCount` `VkPhysicalDevice` handles

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`

To query general properties of physical devices once enumerated, call:

```
void vkGetPhysicalDeviceProperties(  
    VkPhysicalDevice                physicalDevice,  
    VkPhysicalDeviceProperties*    pProperties);
```

- `physicalDevice` is the handle to the physical device whose properties will be queried.
- `pProperties` is a pointer to a `VkPhysicalDeviceProperties` structure in which properties are returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pProperties` **must** be a valid pointer to a `VkPhysicalDeviceProperties` structure

The `VkPhysicalDeviceProperties` structure is defined as:

```

typedef struct VkPhysicalDeviceProperties {
    uint32_t                                apiVersion;
    uint32_t                                driverVersion;
    uint32_t                                vendorID;
    uint32_t                                deviceID;
    VkPhysicalDeviceType                     deviceType;
    char                                     deviceName[VK_MAX_PHYSICAL_DEVICE_NAME_SIZE];
    uint8_t                                  pipelineCacheUUID[VK_UUID_SIZE];
    VkPhysicalDeviceLimits                  limits;
    VkPhysicalDeviceSparseProperties        sparseProperties;
} VkPhysicalDeviceProperties;

```

- **apiVersion** is the version of Vulkan supported by the device, encoded as described in [Version Numbers](#).
- **driverVersion** is the vendor-specified version of the driver.
- **vendorID** is a unique identifier for the *vendor* (see below) of the physical device.
- **deviceID** is a unique identifier for the physical device among devices available from the vendor.
- **deviceType** is a [VkPhysicalDeviceType](#) specifying the type of device.
- **deviceName** is an array of [VK\\_MAX\\_PHYSICAL\\_DEVICE\\_NAME\\_SIZE](#) **char** containing a null-terminated UTF-8 string which is the name of the device.
- **pipelineCacheUUID** is an array of [VK\\_UUID\\_SIZE](#) **uint8\_t** values representing a universally unique identifier for the device.
- **limits** is the [VkPhysicalDeviceLimits](#) structure specifying device-specific limits of the physical device. See [Limits](#) for details.
- **sparseProperties** is the [VkPhysicalDeviceSparseProperties](#) structure specifying various sparse related properties of the physical device. See [Sparse Properties](#) for details.

*Note*

The value of **apiVersion** **may** be different than the version returned by [vkEnumerateInstanceVersion](#); either higher or lower. In such cases, the application **must** not use functionality that exceeds the version of Vulkan associated with a given object. The **pApiVersion** parameter returned by [vkEnumerateInstanceVersion](#) is the version associated with a [VkInstance](#) and its children, except for a [VkPhysicalDevice](#) and its children. **VkPhysicalDeviceProperties::apiVersion** is the version associated with a [VkPhysicalDevice](#) and its children.



The **vendorID** and **deviceID** fields are provided to allow applications to adapt to device characteristics that are not adequately exposed by other Vulkan queries.

*Note*



These **may** include performance profiles, hardware errata, or other characteristics.

The *vendor* identified by `vendorID` is the entity responsible for the most salient characteristics of the underlying implementation of the `VkPhysicalDevice` being queried.

*Note*



For example, in the case of a discrete GPU implementation, this **should** be the GPU chipset vendor. In the case of a hardware accelerator integrated into a system-on-chip (SoC), this **should** be the supplier of the silicon IP used to create the accelerator.

If the vendor has a [PCI vendor ID](#), the low 16 bits of `vendorID` **must** contain that PCI vendor ID, and the remaining bits **must** be set to zero. Otherwise, the value returned **must** be a valid Khronos vendor ID, obtained as described in the [Vulkan Documentation and Extensions: Procedures and Conventions](#) document in the section “Registering a Vendor ID with Khronos”. Khronos vendor IDs are allocated starting at 0x10000, to distinguish them from the PCI vendor ID namespace. Khronos vendor IDs are symbolically defined in the `VkVendorId` type.

The vendor is also responsible for the value returned in `deviceID`. If the implementation is driven primarily by a [PCI device](#) with a [PCI device ID](#), the low 16 bits of `deviceID` **must** contain that PCI device ID, and the remaining bits **must** be set to zero. Otherwise, the choice of what values to return **may** be dictated by operating system or platform policies - but **should** uniquely identify both the device version and any major configuration options (for example, core count in the case of multicore devices).

*Note*



The same device ID **should** be used for all physical implementations of that device version and configuration. For example, all uses of a specific silicon IP GPU version and configuration **should** use the same device ID, even if those uses occur in different SoCs.

Khronos vendor IDs which **may** be returned in `VkPhysicalDeviceProperties::vendorID` are:

```
typedef enum VkVendorId {
    VK_VENDOR_ID_VIV = 0x10001,
    VK_VENDOR_ID_VSI = 0x10002,
    VK_VENDOR_ID_KAZAN = 0x10003,
    VK_VENDOR_ID_MAX_ENUM = 0x7FFFFFFF
} VkVendorId;
```

*Note*



Khronos vendor IDs may be allocated by vendors at any time. Only the latest canonical versions of this Specification, of the corresponding `vk.xml` API Registry, and of the corresponding `vulkan_core.h` header file **must** contain all reserved Khronos vendor IDs.

Only Khronos vendor IDs are given symbolic names at present. PCI vendor IDs returned by the implementation can be looked up in the PCI-SIG database.

The physical device types which **may** be returned in `VkPhysicalDeviceProperties::deviceType` are:

```
typedef enum VkPhysicalDeviceType {
    VK_PHYSICAL_DEVICE_TYPE_OTHER = 0,
    VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU = 1,
    VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU = 2,
    VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU = 3,
    VK_PHYSICAL_DEVICE_TYPE_CPU = 4,
    VK_PHYSICAL_DEVICE_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkPhysicalDeviceType;
```

- `VK_PHYSICAL_DEVICE_TYPE_OTHER` - the device does not match any other available types.
- `VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU` - the device is typically one embedded in or tightly coupled with the host.
- `VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU` - the device is typically a separate processor connected to the host via an interlink.
- `VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU` - the device is typically a virtual node in a virtualization environment.
- `VK_PHYSICAL_DEVICE_TYPE_CPU` - the device is typically running on the same processors as the host.

The physical device type is advertised for informational purposes only, and does not directly affect the operation of the system. However, the device type **may** correlate with other advertised properties or capabilities of the system, such as how many memory heaps there are.

To query general properties of physical devices once enumerated, call:

```
void vkGetPhysicalDeviceProperties2(
    VkPhysicalDevice                      physicalDevice,
    VkPhysicalDeviceProperties2*          pProperties);
```

or the equivalent command

```
void vkGetPhysicalDeviceProperties2KHR(
    VkPhysicalDevice                      physicalDevice,
    VkPhysicalDeviceProperties2*          pProperties);
```

- `physicalDevice` is the handle to the physical device whose properties will be queried.
- `pProperties` is a pointer to a `VkPhysicalDeviceProperties2` structure in which properties are returned.

Each structure in `pProperties` and its `pNext` chain contain members corresponding to properties or implementation-dependent limits. `vkGetPhysicalDeviceProperties2` writes each member to a value indicating the value of that property or limit.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pProperties` **must** be a valid pointer to a `VkPhysicalDeviceProperties2` structure

The `VkPhysicalDeviceProperties2` structure is defined as:

```
typedef struct VkPhysicalDeviceProperties2 {  
    VkStructureType          sType;  
    void*                   pNext;  
    VkPhysicalDeviceProperties properties;  
} VkPhysicalDeviceProperties2;
```

or the equivalent

```
typedef VkPhysicalDeviceProperties2 VkPhysicalDeviceProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `properties` is a `VkPhysicalDeviceProperties` structure describing properties of the physical device. This structure is written with the same values as if it were written by `vkGetPhysicalDeviceProperties`.

The `pNext` chain of this structure is used to extend the structure with properties defined by extensions.

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2`
- Each **pNext** member of any structure (including this one) in the **pNext** chain must be either `NULL` or a pointer to a valid instance of `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT`, `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`, `VkPhysicalDeviceCooperativeMatrixPropertiesNV`, `VkPhysicalDeviceDepthStencilResolvePropertiesKHR`, `VkPhysicalDeviceDescriptorIndexingPropertiesEXT`, `VkPhysicalDeviceDiscardRectanglePropertiesEXT`, `VkPhysicalDeviceDriverPropertiesKHR`, `VkPhysicalDeviceExternalMemoryHostPropertiesEXT`, `VkPhysicalDeviceFloatControlsPropertiesKHR`, `VkPhysicalDeviceFragmentDensityMapPropertiesEXT`, `VkPhysicalDeviceIDProperties`, `VkPhysicalDeviceInlineUniformBlockPropertiesEXT`, `VkPhysicalDeviceLineRasterizationPropertiesEXT`, `VkPhysicalDeviceMaintenance3Properties`, `VkPhysicalDeviceMeshShaderPropertiesNV`, `VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX`, `VkPhysicalDeviceMultiviewProperties`, `VkPhysicalDevicePCIBusInfoPropertiesEXT`, `VkPhysicalDevicePerformanceQueryPropertiesKHR`, `VkPhysicalDevicePointClippingProperties`, `VkPhysicalDeviceProtectedMemoryProperties`, `VkPhysicalDevicePushDescriptorPropertiesKHR`, `VkPhysicalDeviceRayTracingPropertiesNV`, `VkPhysicalDeviceSampleLocationsPropertiesEXT`, `VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT`, `VkPhysicalDeviceShaderCoreProperties2AMD`, `VkPhysicalDeviceShaderCorePropertiesAMD`, `VkPhysicalDeviceShaderSMBuiltinsPropertiesNV`, `VkPhysicalDeviceShadingRateImagePropertiesNV`, `VkPhysicalDeviceSubgroupProperties`, `VkPhysicalDeviceSubgroupSizeControlPropertiesEXT`, `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT`, `VkPhysicalDeviceTimelineSemaphorePropertiesKHR`, `VkPhysicalDeviceTransformFeedbackPropertiesEXT`, `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT` or
- Each **sType** member in the **pNext** chain must be unique

To query the UUID and LUID of a device, add `VkPhysicalDeviceIDProperties` to the **pNext** chain of the `VkPhysicalDeviceProperties2` structure. The `VkPhysicalDeviceIDProperties` structure is defined as:

```

typedef struct VkPhysicalDeviceIDProperties {
    VkStructureType sType;
    void* pNext;
    uint8_t deviceUUID[VK_UUID_SIZE];
    uint8_t driverUUID[VK_UUID_SIZE];
    uint8_t deviceLUID[VK_LUID_SIZE];
    uint32_t deviceNodeMask;
    VkBool32 deviceLUIDValid;
} VkPhysicalDeviceIDProperties;

```

or the equivalent

```
typedef VkPhysicalDeviceIDProperties VkPhysicalDeviceIDPropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **deviceUUID** is an array of **VK\_UUID\_SIZE** **uint8\_t** values representing a universally unique identifier for the device.
- **driverUUID** is an array of **VK\_UUID\_SIZE** **uint8\_t** values representing a universally unique identifier for the driver build in use by the device.
- **deviceLUID** is an array of **VK\_LUID\_SIZE** **uint8\_t** values representing a locally unique identifier for the device.
- **deviceNodeMask** is a **uint32\_t** bitfield identifying the node within a linked device adapter corresponding to the device.
- **deviceLUIDValid** is a boolean value that will be **VK\_TRUE** if **deviceLUID** contains a valid LUID and **deviceNodeMask** contains a valid node mask, and **VK\_FALSE** if they do not.

**deviceUUID** **must** be immutable for a given device across instances, processes, driver APIs, driver versions, and system reboots.

Applications **can** compare the **driverUUID** value across instance and process boundaries, and **can** make similar queries in external APIs to determine whether they are capable of sharing memory objects and resources using them with the device.

**deviceUUID** and/or **driverUUID** **must** be used to determine whether a particular external object can be shared between driver components, where such a restriction exists as defined in the compatibility table for the particular object type:

- [External memory handle types compatibility](#)
- [External semaphore handle types compatibility](#)
- [External fence handle types compatibility](#)

If **deviceLUIDValid** is **VK\_FALSE**, the values of **deviceLUID** and **deviceNodeMask** are undefined. If **deviceLUIDValid** is **VK\_TRUE** and Vulkan is running on the Windows operating system, the contents of

`deviceLUID` can be cast to an `LUID` object and **must** be equal to the locally unique identifier of a `IDXGIAdapter1` object that corresponds to `physicalDevice`. If `deviceLUIDValid` is `VK_TRUE`, `deviceNodeMask` **must** contain exactly one bit. If Vulkan is running on an operating system that supports the Direct3D 12 API and `physicalDevice` corresponds to an individual device in a linked device adapter, `deviceNodeMask` identifies the Direct3D 12 node corresponding to `physicalDevice`. Otherwise, `deviceNodeMask` **must** be 1.

*Note*



Although they have identical descriptions, `VkPhysicalDeviceIDProperties::deviceUUID` may differ from `VkPhysicalDeviceProperties2::pipelineCacheUUID`. The former is intended to identify and correlate devices across API and driver boundaries, while the latter is used to identify a compatible device and driver combination to use when serializing and de-serializing pipeline state.

*Note*



While `VkPhysicalDeviceIDProperties::deviceUUID` is specified to remain consistent across driver versions and system reboots, it is not intended to be usable as a serializable persistent identifier for a device. It may change when a device is physically added to, removed from, or moved to a different connector in a system while that system is powered down. Further, there is no reasonable way to verify with conformance testing that a given device retains the same UUID in a given system across all driver versions supported in that system. While implementations should make every effort to report consistent device UUIDs across driver versions, applications should avoid relying on the persistence of this value for uses other than identifying compatible devices for external object sharing purposes.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES`

To query the properties of the driver corresponding to a physical device, add `VkPhysicalDeviceDriverPropertiesKHR` to the `pNext` chain of the `VkPhysicalDeviceProperties2` structure. The `VkPhysicalDeviceDriverPropertiesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceDriverPropertiesKHR {
    VkStructureType           sType;
    void*                     pNext;
    VkDriverIdKHR             driverID;
    char                      driverName[VK_MAX_DRIVER_NAME_SIZE_KHR];
    char                      driverInfo[VK_MAX_DRIVER_INFO_SIZE_KHR];
    VkConformanceVersionKHR   conformanceVersion;
} VkPhysicalDeviceDriverPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension specific structure.

- `driverID` is a unique identifier for the driver of the physical device.
- `driverName` is an array of `VK_MAX_DRIVER_NAME_SIZE_KHR` `char` containing a null-terminated UTF-8 string which is the name of the driver.
- `driverInfo` is an array of `VK_MAX_DRIVER_INFO_SIZE_KHR` `char` containing a null-terminated UTF-8 string with additional information about the driver.
- `conformanceVersion` is the version of the Vulkan conformance test this driver is conformant against (see [VkConformanceVersionKHR](#)).

`driverID` **must** be immutable for a given driver across instances, processes, driver versions, and system reboots.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES_KHR`

Khronos driver IDs which **may** be returned in [VkPhysicalDeviceDriverPropertiesKHR::driverID](#) are:

```
typedef enum VkDriverIdKHR {
    VK_DRIVER_ID_AMD_PROPRIETARY_KHR = 1,
    VK_DRIVER_ID_AMD_OPEN_SOURCE_KHR = 2,
    VK_DRIVER_ID_MESA_RADV_KHR = 3,
    VK_DRIVER_ID_NVIDIA_PROPRIETARY_KHR = 4,
    VK_DRIVER_ID_INTEL_PROPRIETARY_WINDOWS_KHR = 5,
    VK_DRIVER_ID_INTEL_OPEN_SOURCE_MESA_KHR = 6,
    VK_DRIVER_ID_IMAGINATION_PROPRIETARY_KHR = 7,
    VK_DRIVER_ID_QUALCOMM_PROPRIETARY_KHR = 8,
    VK_DRIVER_ID_ARM_PROPRIETARY_KHR = 9,
    VK_DRIVER_ID_GOOGLE_SWIFTSHADER_KHR = 10,
    VK_DRIVER_ID_GGP_PROPRIETARY_KHR = 11,
    VK_DRIVER_ID_BROADCOM_PROPRIETARY_KHR = 12,
    VK_DRIVER_ID_MAX_ENUM_KHR = 0x7FFFFFFF
} VkDriverIdKHR;
```

### Note

Khronos driver IDs may be allocated by vendors at any time. There may be multiple driver IDs for the same vendor, representing different drivers (for e.g. different platforms, proprietary or open source, etc.). Only the latest canonical versions of this Specification, of the corresponding `vk.xml` API Registry, and of the corresponding `vulkan_core.h` header file **must** contain all reserved Khronos driver IDs.



Only driver IDs registered with Khronos are given symbolic names. There **may** be unregistered driver IDs returned.

The conformance test suite version an implementation is compliant with is described with an instance of the `VkConformanceVersionKHR` structure. The `VkConformanceVersionKHR` structure is defined

as:

```
typedef struct VkConformanceVersionKHR {
    uint8_t    major;
    uint8_t    minor;
    uint8_t    subminor;
    uint8_t    patch;
} VkConformanceVersionKHR;
```

- **major** is the major version number of the conformance test suite.
- **minor** is the minor version number of the conformance test suite.
- **subminor** is the subminor version number of the conformance test suite.
- **patch** is the patch version number of the conformance test suite.

To query the PCI bus information of a physical device, add [VkPhysicalDevicePCIBusInfoPropertiesEXT](#) to the **pNext** chain of the [VkPhysicalDeviceProperties2](#) structure. The [VkPhysicalDevicePCIBusInfoPropertiesEXT](#) structure is defined as:

```
typedef struct VkPhysicalDevicePCIBusInfoPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            pciDomain;
    uint32_t            pciBus;
    uint32_t            pciDevice;
    uint32_t            pciFunction;
} VkPhysicalDevicePCIBusInfoPropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **pciDomain** is the PCI bus domain.
- **pciBus** is the PCI bus identifier.
- **pciDevice** is the PCI device identifier.
- **pciFunction** is the PCI device function identifier.

## Valid Usage (Implicit)

- **sType** must be **VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_PCI\_BUS\_INFO\_PROPERTIES\_EXT**

To query properties of queues available on a physical device, call:

```
void vkGetPhysicalDeviceQueueFamilyProperties(
    VkPhysicalDevice physicalDevice,
    uint32_t* pQueueFamilyPropertyCount,
    VkQueueFamilyProperties* pQueueFamilyProperties);
```

- `physicalDevice` is the handle to the physical device whose properties will be queried.
- `pQueueFamilyPropertyCount` is a pointer to an integer related to the number of queue families available or queried, as described below.
- `pQueueFamilyProperties` is either `NULL` or a pointer to an array of `VkQueueFamilyProperties` structures.

If `pQueueFamilyProperties` is `NULL`, then the number of queue families available is returned in `pQueueFamilyPropertyCount`. Implementations **must** support at least one queue family. Otherwise, `pQueueFamilyPropertyCount` **must** point to a variable set by the user to the number of elements in the `pQueueFamilyProperties` array, and on return the variable is overwritten with the number of structures actually written to `pQueueFamilyProperties`. If `pQueueFamilyPropertyCount` is less than the number of queue families available, at most `pQueueFamilyPropertyCount` structures will be written.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pQueueFamilyPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pQueueFamilyPropertyCount` is not `0`, and `pQueueFamilyProperties` is not `NULL`, `pQueueFamilyProperties` **must** be a valid pointer to an array of `pQueueFamilyPropertyCount` `VkQueueFamilyProperties` structures

The `VkQueueFamilyProperties` structure is defined as:

```
typedef struct VkQueueFamilyProperties {
    VkQueueFlags    queueFlags;
    uint32_t        queueCount;
    uint32_t        timestampValidBits;
    VkExtent3D      minImageTransferGranularity;
} VkQueueFamilyProperties;
```

- `queueFlags` is a bitmask of `VkQueueFlagBits` indicating capabilities of the queues in this queue family.
- `queueCount` is the unsigned integer count of queues in this queue family. Each queue family **must** support at least one queue.
- `timestampValidBits` is the unsigned integer count of meaningful bits in the timestamps written via `vkCmdWriteTimestamp`. The valid range for the count is 36..64 bits, or a value of 0, indicating no support for timestamps. Bits outside the valid range are guaranteed to be zeros.
- `minImageTransferGranularity` is the minimum granularity supported for image transfer

operations on the queues in this queue family.

The value returned in `minImageTransferGranularity` has a unit of compressed texel blocks for images having a block-compressed format, and a unit of texels otherwise.

Possible values of `minImageTransferGranularity` are:

- (0,0,0) which indicates that only whole mip levels **must** be transferred using the image transfer operations on the corresponding queues. In this case, the following restrictions apply to all offset and extent parameters of image transfer operations:
  - The `x`, `y`, and `z` members of a `VkOffset3D` parameter **must** always be zero.
  - The `width`, `height`, and `depth` members of a `VkExtent3D` parameter **must** always match the width, height, and depth of the image subresource corresponding to the parameter, respectively.
- ( $A_x$ ,  $A_y$ ,  $A_z$ ) where  $A_x$ ,  $A_y$ , and  $A_z$  are all integer powers of two. In this case the following restrictions apply to all image transfer operations:
  - `x`, `y`, and `z` of a `VkOffset3D` parameter **must** be integer multiples of  $A_x$ ,  $A_y$ , and  $A_z$ , respectively.
  - `width` of a `VkExtent3D` parameter **must** be an integer multiple of  $A_x$ , or else `x + width` **must** equal the width of the image subresource corresponding to the parameter.
  - `height` of a `VkExtent3D` parameter **must** be an integer multiple of  $A_y$ , or else `y + height` **must** equal the height of the image subresource corresponding to the parameter.
  - `depth` of a `VkExtent3D` parameter **must** be an integer multiple of  $A_z$ , or else `z + depth` **must** equal the depth of the image subresource corresponding to the parameter.
  - If the format of the image corresponding to the parameters is one of the block-compressed formats then for the purposes of the above calculations the granularity **must** be scaled up by the compressed texel block dimensions.

Queues supporting graphics and/or compute operations **must** report (1,1,1) in `minImageTransferGranularity`, meaning that there are no additional restrictions on the granularity of image transfer operations for these queues. Other queues supporting image transfer operations are only **required** to support whole mip level transfers, thus `minImageTransferGranularity` for queues belonging to such queue families **may** be (0,0,0).

The [Device Memory](#) section describes memory properties queried from the physical device.

For physical device feature queries see the [Features](#) chapter.

Bits which **may** be set in `VkQueueFamilyProperties::queueFlags` indicating capabilities of queues in a queue family are:

```
typedef enum VkQueueFlagBits {
    VK_QUEUE_GRAPHICS_BIT = 0x00000001,
    VK_QUEUE_COMPUTE_BIT = 0x00000002,
    VK_QUEUE_TRANSFER_BIT = 0x00000004,
    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,
    VK_QUEUE_PROTECTED_BIT = 0x00000010,
    VK_QUEUE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkQueueFlagBits;
```

- `VK_QUEUE_GRAPHICS_BIT` specifies that queues in this queue family support graphics operations.
- `VK_QUEUE_COMPUTE_BIT` specifies that queues in this queue family support compute operations.
- `VK_QUEUE_TRANSFER_BIT` specifies that queues in this queue family support transfer operations.
- `VK_QUEUE_SPARSE_BINDING_BIT` specifies that queues in this queue family support sparse memory management operations (see [Sparse Resources](#)). If any of the sparse resource features are enabled, then at least one queue family **must** support this bit.
- if `VK_QUEUE_PROTECTED_BIT` is set, then the queues in this queue family support the `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT` bit. (see [Protected Memory](#)). If the protected memory physical device feature is supported, then at least one queue family of at least one physical device exposed by the implementation **must** support this bit.

If an implementation exposes any queue family that supports graphics operations, at least one queue family of at least one physical device exposed by the implementation **must** support both graphics and compute operations.

Furthermore, if the protected memory physical device feature is supported, then at least one queue family of at least one physical device exposed by the implementation **must** support graphics operations, compute operations, and protected memory operations.

*Note*



All commands that are allowed on a queue that supports transfer operations are also allowed on a queue that supports either graphics or compute operations. Thus, if the capabilities of a queue family include `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`, then reporting the `VK_QUEUE_TRANSFER_BIT` capability separately for that queue family is **optional**.

For further details see [Queues](#).

```
typedef VkFlags VkQueueFlags;
```

`VkQueueFlags` is a bitmask type for setting a mask of zero or more `VkQueueFlagBits`.

To query properties of queues available on a physical device, call:

```
void vkGetPhysicalDeviceQueueFamilyProperties2(
    VkPhysicalDevice                          physicalDevice,
    uint32_t*                                 pQueueFamilyPropertyCount,
    VkQueueFamilyProperties2*                  pQueueFamilyProperties);
```

or the equivalent command

```
void vkGetPhysicalDeviceQueueFamilyProperties2KHR(
    VkPhysicalDevice                          physicalDevice,
    uint32_t*                                pQueueFamilyPropertyCount,
    VkQueueFamilyProperties2*                 pQueueFamilyProperties);
```

- **physicalDevice** is the handle to the physical device whose properties will be queried.
- **pQueueFamilyPropertyCount** is a pointer to an integer related to the number of queue families available or queried, as described in [vkGetPhysicalDeviceQueueFamilyProperties](#).
- **pQueueFamilyProperties** is either **NULL** or a pointer to an array of [VkQueueFamilyProperties2](#) structures.

[vkGetPhysicalDeviceQueueFamilyProperties2](#) behaves similarly to [vkGetPhysicalDeviceQueueFamilyProperties](#), with the ability to return extended information in a **pNext** chain of output structures.

## Valid Usage (Implicit)

- **physicalDevice** **must** be a valid [VkPhysicalDevice](#) handle
- **pQueueFamilyPropertyCount** **must** be a valid pointer to a [uint32\\_t](#) value
- If the value referenced by **pQueueFamilyPropertyCount** is not **0**, and **pQueueFamilyProperties** is not **NULL**, **pQueueFamilyProperties** **must** be a valid pointer to an array of **pQueueFamilyPropertyCount** [VkQueueFamilyProperties2](#) structures

The [VkQueueFamilyProperties2](#) structure is defined as:

```
typedef struct VkQueueFamilyProperties2 {
    VkStructureType          sType;
    void*                   pNext;
    VkQueueFamilyProperties queueFamilyProperties;
} VkQueueFamilyProperties2;
```

or the equivalent

```
typedef VkQueueFamilyProperties2 VkQueueFamilyProperties2KHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **queueFamilyProperties** is a [VkQueueFamilyProperties](#) structure which is populated with the same values as in [vkGetPhysicalDeviceQueueFamilyProperties](#).

## Valid Usage (Implicit)

- **sType** **must** be [VK\\_STRUCTURE\\_TYPE\\_QUEUE\\_FAMILY\\_PROPERTIES\\_2](#)
- **pNext** **must** be **NULL** or a pointer to a valid instance of [VkQueueFamilyCheckpointPropertiesNV](#)

Additional queue family information can be queried by setting [VkQueueFamilyProperties2::pNext](#) to point to an instance of the [VkQueueFamilyCheckpointPropertiesNV](#) structure.

The [VkQueueFamilyCheckpointPropertiesNV](#) structure is defined as:

```
typedef struct VkQueueFamilyCheckpointPropertiesNV {
    VkStructureType          sType;
    void*                    pNext;
    VkPipelineStageFlags     checkpointExecutionStageMask;
} VkQueueFamilyCheckpointPropertiesNV;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **checkpointExecutionStageMask** is a mask indicating which pipeline stages the implementation can execute checkpoint markers in.

## Valid Usage (Implicit)

- **sType** **must** be [VK\\_STRUCTURE\\_TYPE\\_QUEUE\\_FAMILY\\_CHECKPOINT\\_PROPERTIES\\_NV](#)

To enumerate the performance query counters available on a queue family of a physical device, call:

```
VkResult vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(
    VkPhysicalDevice                  physicalDevice,
    uint32_t                          queueFamilyIndex,
    uint32_t*                         pCounterCount,
    VkPerformanceCounterKHR*          pCounters,
    VkPerformanceCounterDescriptionKHR* pCounterDescriptions);
```

- **physicalDevice** is the handle to the physical device whose queue family performance query counter properties will be queried

- `queueFamilyIndex` is the index into the queue family of the physical device we want to get properties for
- `pCounterCount` is a pointer to an integer related to the number of counters available or queried, as described below
- `pCounters` is either `NULL` or a pointer to an array of `VkPerformanceCounterKHR` structures
- `pCounterDescriptions` is either `NULL` or a pointer to an array of `VkPerformanceCounterDescriptionKHR` structures

If `pCounters` is `NULL` and `pCounterDescriptions` is `NULL`, then the number of counters available is returned in `pCounterCount`. Otherwise, `pCounterCount` **must** point to a variable set by the user to the number of elements in the `pCounters`, `pCounterDescriptions`, or both arrays and on return the variable is overwritten with the number of structures actually written out. If `pCounterCount` is less than the number of counters available, at most `pCounterCount` structures will be written and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pCounterCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pCounterCount` is not `0`, and `pCounters` is not `NULL`, `pCounters` **must** be a valid pointer to an array of `pCounterCount` `VkPerformanceCounterKHR` structures
- If the value referenced by `pCounterCount` is not `0`, and `pCounterDescriptions` is not `NULL`, `pCounterDescriptions` **must** be a valid pointer to an array of `pCounterCount` `VkPerformanceCounterDescriptionKHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`

The `VkPerformanceCounterKHR` structure is defined as:

```

typedef struct VkPerformanceCounterKHR {
    VkStructureType          sType;
    const void*             pNext;
    VkPerformanceCounterUnitKHR unit;
    VkPerformanceCounterScopeKHR scope;
    VkPerformanceCounterStorageKHR storage;
    uint8_t                  uuid[VK_UUID_SIZE];
} VkPerformanceCounterKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **unit** is a **VkPerformanceCounterUnitKHR** specifying the unit that the counter data will record
- **scope** is a **VkPerformanceCounterScopeKHR** specifying the scope that the counter belongs to
- **storage** is a **VkPerformanceCounterStorageKHR** specifying the storage type that the counter's data uses
- **uuid** is an array of size **VK\_UUID\_SIZE**, containing 8-bit values that represent a universally unique identifier for the counter of the physical device.

### Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_PERFORMANCE\_COUNTER\_KHR**
- **pNext** **must** be **NULL**

Performance counters have an associated unit. This unit describes how to interpret the performance counter result.

The performance counter unit types which **may** be returned in **VkPerformanceCounterKHR::unit** are:

```

typedef enum VkPerformanceCounterUnitKHR {
    VK_PERFORMANCE_COUNTER_UNIT_GENERIC_KHR = 0,
    VK_PERFORMANCE_COUNTER_UNIT_PERCENTAGE_KHR = 1,
    VK_PERFORMANCE_COUNTER_UNIT_NANOSECONDS_KHR = 2,
    VK_PERFORMANCE_COUNTER_UNIT_BYTES_KHR = 3,
    VK_PERFORMANCE_COUNTER_UNIT_BYTES_PER_SECOND_KHR = 4,
    VK_PERFORMANCE_COUNTER_UNIT_KELVIN_KHR = 5,
    VK_PERFORMANCE_COUNTER_UNIT_WATTS_KHR = 6,
    VK_PERFORMANCE_COUNTER_UNIT_VOLTS_KHR = 7,
    VK_PERFORMANCE_COUNTER_UNIT_AMPS_KHR = 8,
    VK_PERFORMANCE_COUNTER_UNIT_HERTZ_KHR = 9,
    VK_PERFORMANCE_COUNTER_UNIT_CYCLES_KHR = 10,
    VK_PERFORMANCE_COUNTER_UNIT_MAX_ENUM_KHR = 0x7FFFFFFF
} VkPerformanceCounterUnitKHR;

```

- `VK_PERFORMANCE_COUNTER_UNIT_GENERIC_KHR` - the performance counter unit is a generic data point
- `VK_PERFORMANCE_COUNTER_UNIT_PERCENTAGE_KHR` - the performance counter unit is a percentage (%)
- `VK_PERFORMANCE_COUNTER_UNIT_NANOSECONDS_KHR` - the performance counter unit is a value of nanoseconds (ns)
- `VK_PERFORMANCE_COUNTER_UNIT_BYTES_KHR` - the performance counter unit is a value of bytes
- `VK_PERFORMANCE_COUNTER_UNIT_BYTES_PER_SECOND_KHR` - the performance counter unit is a value of bytes/s
- `VK_PERFORMANCE_COUNTER_UNIT_KELVIN_KHR` - the performance counter unit is a temperature reported in Kelvin
- `VK_PERFORMANCE_COUNTER_UNIT_WATTS_KHR` - the performance counter unit is a value of watts (W)
- `VK_PERFORMANCE_COUNTER_UNIT_VOLTS_KHR` - the performance counter unit is a value of volts (V)
- `VK_PERFORMANCE_COUNTER_UNIT_AMPS_KHR` - the performance counter unit is a value of amps (A)
- `VK_PERFORMANCE_COUNTER_UNIT_HERTZ_KHR` - the performance counter unit is a value of hertz (Hz)
- `VK_PERFORMANCE_COUNTER_UNIT_CYCLES_KHR` - the performance counter unit is a value of cycles

Performance counters have an associated scope. This scope describes the granularity of a performance counter.

The performance counter scope types which **may** be returned in `VkPerformanceCounterKHR::scope` are:

```
typedef enum VkPerformanceCounterScopeKHR {
    VK_QUERY_SCOPE_COMMAND_BUFFER_KHR = 0,
    VK_QUERY_SCOPE_RENDER_PASS_KHR = 1,
    VK_QUERY_SCOPE_COMMAND_KHR = 2,
    VK_PERFORMANCE_COUNTER_SCOPE_MAX_ENUM_KHR = 0x7FFFFFFF
} VkPerformanceCounterScopeKHR;
```

- `VK_QUERY_SCOPE_COMMAND_BUFFER_KHR` - the performance counter scope is a single complete command buffer
- `VK_QUERY_SCOPE_RENDER_PASS_KHR` - the performance counter scope is zero or more complete render passes. The performance query containing the performance counter **must** begin and end outside a render pass instance
- `VK_QUERY_SCOPE_COMMAND_KHR` - the performance counter scope is zero or more commands

Performance counters have an associated storage. This storage describes the payload of a counter result.

The performance counter storage types which **may** be returned in `VkPerformanceCounterKHR::storage` are:

```

typedef enum VkPerformanceCounterStorageKHR {
    VK_PERFORMANCE_COUNTER_STORAGE_INT32_KHR = 0,
    VK_PERFORMANCE_COUNTER_STORAGE_INT64_KHR = 1,
    VK_PERFORMANCE_COUNTER_STORAGE_UINT32_KHR = 2,
    VK_PERFORMANCE_COUNTER_STORAGE_UINT64_KHR = 3,
    VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32_KHR = 4,
    VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64_KHR = 5,
    VK_PERFORMANCE_COUNTER_STORAGE_MAX_ENUM_KHR = 0x7FFFFFFF
} VkPerformanceCounterStorageKHR;

```

- **VK\_PERFORMANCE\_COUNTER\_STORAGE\_INT32\_KHR** - the performance counter storage is a 32-bit signed integer
- **VK\_PERFORMANCE\_COUNTER\_STORAGE\_INT64\_KHR** - the performance counter storage is a 64-bit signed integer
- **VK\_PERFORMANCE\_COUNTER\_STORAGE\_UINT32\_KHR** - the performance counter storage is a 32-bit unsigned integer
- **VK\_PERFORMANCE\_COUNTER\_STORAGE\_UINT64\_KHR** - the performance counter storage is a 64-bit unsigned integer
- **VK\_PERFORMANCE\_COUNTER\_STORAGE\_FLOAT32\_KHR** - the performance counter storage is a 32-bit floating-point
- **VK\_PERFORMANCE\_COUNTER\_STORAGE\_FLOAT64\_KHR** - the performance counter storage is a 64-bit floating-point

The **VkPerformanceCounterDescriptionKHR** structure is defined as:

```

typedef struct VkPerformanceCounterDescriptionKHR {
    VkStructureType                      sType;
    const void*                         pNext;
    VkPerformanceCounterDescriptionFlagsKHR flags;
    char                                name[VK_MAX_DESCRIPTION_SIZE];
    char                                category[VK_MAX_DESCRIPTION_SIZE];
    char                                description[VK_MAX_DESCRIPTION_SIZE];
} VkPerformanceCounterDescriptionKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is a bitmask of **VkPerformanceCounterDescriptionFlagBitsKHR** indicating the usage behavior for the counter
- **name** is an array of size **VK\_MAX\_DESCRIPTION\_SIZE**, containing a null-terminated UTF-8 string specifying the name of the counter
- **category** is an array of size **VK\_MAX\_DESCRIPTION\_SIZE**, containing a null-terminated UTF-8 string specifying the category of the counter
- **description** is an array of size **VK\_MAX\_DESCRIPTION\_SIZE**, containing a null-terminated UTF-8

string specifying the description of the counter

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR`
- `pNext` **must** be `NULL`

Bits which **can** be set in `VkPerformanceCounterDescriptionKHR::flags` to specify usage behavior for a command pool are:

```
typedef enum VkPerformanceCounterDescriptionFlagBitsKHR {
    VK_PERFORMANCE_COUNTER_DESCRIPTION_PERFORMANCE_IMPACTING_KHR = 0x00000001,
    VK_PERFORMANCE_COUNTER_DESCRIPTION_CONCURRENTLY_IMPACTED_KHR = 0x00000002,
    VK_PERFORMANCE_COUNTER_DESCRIPTION_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkPerformanceCounterDescriptionFlagBitsKHR;
```

- `VK_PERFORMANCE_COUNTER_DESCRIPTION_PERFORMANCE_IMPACTING_KHR` specifies that recording the counter **may** have a noticeable performance impact
- `VK_PERFORMANCE_COUNTER_DESCRIPTION_CONCURRENTLY_IMPACTED_KHR` specifies that concurrently recording the counter while other submitted command buffers are running **may** impact the accuracy of the recording

```
typedef VkFlags VkPerformanceCounterDescriptionFlagsKHR;
```

`VkPerformanceCounterDescriptionFlagsKHR` is a bitmask type for setting a mask of zero or more `VkPerformanceCounterDescriptionFlagBitsKHR`.

## 4.2. Devices

Device objects represent logical connections to physical devices. Each device exposes a number of *queue families* each having one or more *queues*. All queues in a queue family support the same operations.

As described in [Physical Devices](#), a Vulkan application will first query for all physical devices in a system. Each physical device **can** then be queried for its capabilities, including its queue and queue family properties. Once an acceptable physical device is identified, an application will create a corresponding logical device. An application **must** create a separate logical device for each physical device it will use. The created logical device is then the primary interface to the physical device.

How to enumerate the physical devices in a system and query those physical devices for their queue family properties is described in the [Physical Device Enumeration](#) section above.

A single logical device **can** also be created from multiple physical devices, if those physical devices belong to the same device group. A *device group* is a set of physical devices that support accessing each other's memory and recording a single command buffer that **can** be executed on all the

physical devices. Device groups are enumerated by calling [vkEnumeratePhysicalDeviceGroups](#), and a logical device is created from a subset of the physical devices in a device group by passing the physical devices through [VkDeviceGroupCreateInfo](#). For two physical devices to be in the same device group, they **must** support identical extensions, features, and properties.

*Note*

Physical devices in the same device group **must** be so similar because there are no rules for how different features/properties would interact. They **must** return the same values for nearly every invariant [vkGetPhysicalDevice\\*](#) feature, property, capability, etc., but could potentially differ for certain queries based on things like having a different display connected, or different compositor, etc.. The specification does not attempt to enumerate which state is in each category, because such a list would quickly become out of date.



To retrieve a list of the device groups present in the system, call:

```
VkResult vkEnumeratePhysicalDeviceGroups(  
    VkInstance  
    uint32_t*  
    VkPhysicalDeviceGroupProperties*  
        instance,  
        pPhysicalDeviceGroupCount,  
        pPhysicalDeviceGroupProperties);
```

or the equivalent command

```
VkResult vkEnumeratePhysicalDeviceGroupsKHR(  
    VkInstance  
    uint32_t*  
    VkPhysicalDeviceGroupProperties*  
        instance,  
        pPhysicalDeviceGroupCount,  
        pPhysicalDeviceGroupProperties);
```

- `instance` is a handle to a Vulkan instance previously created with [vkCreateInstance](#).
- `pPhysicalDeviceGroupCount` is a pointer to an integer related to the number of device groups available or queried, as described below.
- `pPhysicalDeviceGroupProperties` is either `NULL` or a pointer to an array of [VkPhysicalDeviceGroupProperties](#) structures.

If `pPhysicalDeviceGroupProperties` is `NULL`, then the number of device groups available is returned in `pPhysicalDeviceGroupCount`. Otherwise, `pPhysicalDeviceGroupCount` **must** point to a variable set by the user to the number of elements in the `pPhysicalDeviceGroupProperties` array, and on return the variable is overwritten with the number of structures actually written to `pPhysicalDeviceGroupProperties`. If `pPhysicalDeviceGroupCount` is less than the number of device groups available, at most `pPhysicalDeviceGroupCount` structures will be written. If `pPhysicalDeviceGroupCount` is smaller than the number of device groups available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available device groups were returned.

Every physical device **must** be in exactly one device group.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pPhysicalDeviceGroupCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPhysicalDeviceGroupCount` is not `0`, and `pPhysicalDeviceGroupProperties` is not `NULL`, `pPhysicalDeviceGroupProperties` **must** be a valid pointer to an array of `pPhysicalDeviceGroupCount` `VkPhysicalDeviceGroupProperties` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`

The `VkPhysicalDeviceGroupProperties` structure is defined as:

```
typedef struct VkPhysicalDeviceGroupProperties {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            physicalDeviceCount;
    VkPhysicalDevice    physicalDevices[VK_MAX_DEVICE_GROUP_SIZE];
    VkBool32            subsetAllocation;
} VkPhysicalDeviceGroupProperties;
```

or the equivalent

```
typedef VkPhysicalDeviceGroupProperties VkPhysicalDeviceGroupPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `physicalDeviceCount` is the number of physical devices in the group.
- `physicalDevices` is an array of `VK_MAX_DEVICE_GROUP_SIZE` `VkPhysicalDevice` handles representing all physical devices in the group. The first `physicalDeviceCount` elements of the array will be valid.
- `subsetAllocation` specifies whether logical devices created from the group support allocating device memory on a subset of devices, via the `deviceMask` member of the

`VkMemoryAllocateFlagsInfo`. If this is `VK_FALSE`, then all device memory allocations are made across all physical devices in the group. If `physicalDeviceCount` is 1, then `subsetAllocation` **must** be `VK_FALSE`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES`
- `pNext` **must** be `NULL`

#### 4.2.1. Device Creation

Logical devices are represented by `VkDevice` handles:

```
VK_DEFINE_HANDLE(VkDevice)
```

A logical device is created as a *connection* to a physical device. To create a logical device, call:

```
VkResult vkCreateDevice(  
    VkPhysicalDevice physicalDevice,  
    const VkDeviceCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkDevice* pDevice);
```

- `physicalDevice` **must** be one of the device handles returned from a call to `vkEnumeratePhysicalDevices` (see [Physical Device Enumeration](#)).
- `pCreateInfo` is a pointer to a `VkDeviceCreateInfo` structure containing information about how to create the device.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pDevice` is a pointer to a handle in which the created `VkDevice` is returned.

`vkCreateDevice` verifies that extensions and features requested in the `ppEnabledExtensionNames` and `pEnabledFeatures` members of `pCreateInfo`, respectively, are supported by the implementation. If any requested extension is not supported, `vkCreateDevice` **must** return `VK_ERROR_EXTENSION_NOT_PRESENT`. If any requested feature is not supported, `vkCreateDevice` **must** return `VK_ERROR_FEATURE_NOT_PRESENT`. Support for extensions **can** be checked before creating a device by querying `vkEnumerateDeviceExtensionProperties`. Support for features **can** similarly be checked by querying `vkGetPhysicalDeviceFeatures`.

After verifying and enabling the extensions the `VkDevice` object is created and returned to the application. If a requested extension is only supported by a layer, both the layer and the extension need to be specified at `vkCreateInstance` time for the creation to succeed.

Multiple logical devices **can** be created from the same physical device. Logical device creation **may** fail due to lack of device-specific resources (in addition to the other errors). If that occurs,

`vkCreateDevice` will return `VK_ERROR_TOO_MANY_OBJECTS`.

## Valid Usage

- All required extensions for each extension in the `VkDeviceCreateInfo` `::ppEnabledExtensionNames` list **must** also be present in that list.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkDeviceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pDevice` **must** be a valid pointer to a `VkDevice` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`
- `VK_ERROR_EXTENSION_NOT_PRESENT`
- `VK_ERROR_FEATURE_NOT_PRESENT`
- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_DEVICE_LOST`

The `VkDeviceCreateInfo` structure is defined as:

```
typedef struct VkDeviceCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkDeviceCreateFlags        flags;
    uint32_t                  queueCreateInfoCount;
    const VkDeviceQueueCreateInfo* pQueueCreateInfos;
    uint32_t                  enabledLayerCount;
    const char* const*        ppEnabledLayerNames;
    uint32_t                  enabledExtensionCount;
    const char* const*        ppEnabledExtensionNames;
    const VkPhysicalDeviceFeatures* pEnabledFeatures;
} VkDeviceCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `queueCreateInfoCount` is the unsigned integer size of the `pQueueCreateInfos` array. Refer to the [Queue Creation](#) section below for further details.
- `pQueueCreateInfos` is a pointer to an array of `VkDeviceQueueCreateInfo` structures describing the queues that are requested to be created along with the logical device. Refer to the [Queue Creation](#) section below for further details.
- `enabledLayerCount` is deprecated and ignored.
- `ppEnabledLayerNames` is deprecated and ignored. See [Device Layer Deprecation](#).
- `enabledExtensionCount` is the number of device extensions to enable.
- `ppEnabledExtensionNames` is a pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings containing the names of extensions to enable for the created device. See the [Extensions](#) section for further details.
- `pEnabledFeatures` is `NULL` or a pointer to a `VkPhysicalDeviceFeatures` structure containing boolean indicators of all the features to be enabled. Refer to the [Features](#) section for further details.

## Valid Usage

- The `queueFamilyIndex` member of each element of `pQueueCreateInfos` **must** be unique within `pQueueCreateInfos`, except that two members can share the same `queueFamilyIndex` if one is a protected-capable queue and one is not a protected-capable queue.
- If the `pNext` chain includes a `VkPhysicalDeviceFeatures2` structure, then `pEnabledFeatures` **must** be `NULL`
- `ppEnabledExtensionNames` **must** not contain `VK_AMD_negative_viewport_height`
- `ppEnabledExtensionNames` **must** not contain both `VK_KHR_buffer_device_address` and `VK_EXT_buffer_device_address`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDeviceGroupDeviceCreateInfo`, `VkDeviceMemoryOverallocationCreateInfoAMD`, `VkPhysicalDevice16BitStorageFeatures`, `VkPhysicalDevice8BitStorageFeaturesKHR`, `VkPhysicalDeviceASTCDecodeFeaturesEXT`, `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT`, `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT`, `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR`, `VkPhysicalDeviceCoherentMemoryFeaturesAMD`, `VkPhysicalDeviceComputeShaderDerivativesFeaturesNV`, `VkPhysicalDeviceConditionalRenderingFeaturesEXT`, `VkPhysicalDeviceCooperativeMatrixFeaturesNV`, `VkPhysicalDeviceCornerSampledImageFeaturesNV`, `VkPhysicalDeviceCoverageReductionModeFeaturesNV`, `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV`, `VkPhysicalDeviceDepthClipEnableFeaturesEXT`, `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`, `VkPhysicalDeviceExclusiveScissorFeaturesNV`, `VkPhysicalDeviceFragmentDensityMapFeaturesEXT`, `VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV`, `VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT`, `VkPhysicalDeviceHostQueryResetFeaturesEXT`, `VkPhysicalDeviceImagelessFramebufferFeaturesKHR`, `VkPhysicalDeviceIndexTypeUint8FeaturesEXT`, `VkPhysicalDeviceInlineUniformBlockFeaturesEXT`, `VkPhysicalDeviceLineRasterizationFeaturesEXT`, `VkPhysicalDeviceMemoryPriorityFeaturesEXT`, `VkPhysicalDeviceMeshShaderFeaturesNV`, `VkPhysicalDeviceMultiviewFeatures`, `VkPhysicalDevicePerformanceQueryFeaturesKHR`, `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR`, `VkPhysicalDeviceProtectedMemoryFeatures`, `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV`, `VkPhysicalDeviceSamplerYcbcrConversionFeatures`, `VkPhysicalDeviceScalarBlockLayoutFeaturesEXT`, `VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR`, `VkPhysicalDeviceShaderAtomicInt64FeaturesKHR`, `VkPhysicalDeviceShaderClockFeaturesKHR`, `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT`, `VkPhysicalDeviceShaderDrawParametersFeatures`, `VkPhysicalDeviceShaderFloat16Int8FeaturesKHR`, `VkPhysicalDeviceShaderImageFootprintFeaturesNV`, `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL`, `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV`, `VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR`, `VkPhysicalDeviceShadingRateImageFeaturesNV`, `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT`, `VkPhysicalDeviceFeatures2`,

```
VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT,  
VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT,  
VkPhysicalDeviceTimelineSemaphoreFeaturesKHR,  
VkPhysicalDeviceTransformFeedbackFeaturesEXT,  
VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR,  
VkPhysicalDeviceVariablePointersFeatures,  
VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT,  
VkPhysicalDeviceVulkanMemoryModelFeaturesKHR,  
VkPhysicalDeviceYcbcrImageArraysFeaturesEXT
```

or

- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be `0`
- `pQueueCreateInfos` **must** be a valid pointer to an array of `queueCreateInfoCount` valid `VkDeviceQueueCreateInfo` structures
- If `enabledLayerCount` is not `0`, `ppEnabledLayerNames` **must** be a valid pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings
- If `enabledExtensionCount` is not `0`, `ppEnabledExtensionNames` **must** be a valid pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings
- If `pEnabledFeatures` is not `NULL`, `pEnabledFeatures` **must** be a valid pointer to a valid `VkPhysicalDeviceFeatures` structure
- `queueCreateInfoCount` **must** be greater than `0`

```
typedef VkFlags VkDeviceCreateFlags;
```

`VkDeviceCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

A logical device **can** be created that connects to one or more physical devices by including a `VkDeviceGroupDeviceCreateInfo` structure in the `pNext` chain of `VkDeviceCreateInfo`. The `VkDeviceGroupDeviceCreateInfo` structure is defined as:

```
typedef struct VkDeviceGroupDeviceCreateInfo {  
    VkStructureType          sType;  
    const void*              pNext;  
    uint32_t                 physicalDeviceCount;  
    const VkPhysicalDevice*   pPhysicalDevices;  
} VkDeviceGroupDeviceCreateInfo;
```

or the equivalent

```
typedef VkDeviceGroupDeviceCreateInfo VkDeviceGroupDeviceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `physicalDeviceCount` is the number of elements in the `pPhysicalDevices` array.
- `pPhysicalDevices` is a pointer to an array of physical device handles belonging to the same device group.

The elements of the `pPhysicalDevices` array are an ordered list of the physical devices that the logical device represents. These **must** be a subset of a single device group, and need not be in the same order as they were enumerated. The order of the physical devices in the `pPhysicalDevices` array determines the *device index* of each physical device, with element *i* being assigned a device index of *i*. Certain commands and structures refer to one or more physical devices by using device indices or *device masks* formed using device indices.

A logical device created without using `VkDeviceGroupDeviceCreateInfo`, or with `physicalDeviceCount` equal to zero, is equivalent to a `physicalDeviceCount` of one and `pPhysicalDevices` pointing to the `physicalDevice` parameter to `vkCreateDevice`. In particular, the device index of that physical device is zero.

## Valid Usage

- Each element of `pPhysicalDevices` **must** be unique
- All elements of `pPhysicalDevices` **must** be in the same device group as enumerated by `vkEnumeratePhysicalDeviceGroups`
- If `physicalDeviceCount` is not `0`, the `physicalDevice` parameter of `vkCreateDevice` **must** be an element of `pPhysicalDevices`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO`
- If `physicalDeviceCount` is not `0`, `pPhysicalDevices` **must** be a valid pointer to an array of `physicalDeviceCount` valid `VkPhysicalDevice` handles

To specify whether device memory allocation is allowed beyond the size reported by `VkPhysicalDeviceMemoryProperties`, add a `VkDeviceMemoryOverallocationCreateInfoAMD` structure to the `pNext` chain of the `VkDeviceCreateInfo` structure. If this structure is not specified, it is as if the `VK_MEMORY_OVERALLOCATION_BEHAVIOR_DEFAULT_AMD` value is used.

```
typedef struct VkDeviceMemoryOverallocationCreateInfoAMD {
    VkStructureType           sType;
    const void*               pNext;
    VkMemoryOverallocationBehaviorAMD   overallocationBehavior;
} VkDeviceMemoryOverallocationCreateInfoAMD;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `overallocationBehavior` is the desired overallocation behavior.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_MEMORY_OVERALLOCATION_CREATE_INFO_AMD`
- `overallocationBehavior` **must** be a valid `VkMemoryOverallocationBehaviorAMD` value

Possible values for `VkDeviceMemoryOverallocationCreateInfoAMD::overallocationBehavior` include:

```
typedef enum VkMemoryOverallocationBehaviorAMD {
    VK_MEMORY_OVERALLOCATION_BEHAVIOR_DEFAULT_AMD = 0,
    VK_MEMORY_OVERALLOCATION_BEHAVIOR_ALLOWED_AMD = 1,
    VK_MEMORY_OVERALLOCATION_BEHAVIOR_DISALLOWED_AMD = 2,
    VK_MEMORY_OVERALLOCATION_BEHAVIOR_MAX_ENUM_AMD = 0x7FFFFFFF
} VkMemoryOverallocationBehaviorAMD;
```

- `VK_MEMORY_OVERALLOCATION_BEHAVIOR_DEFAULT_AMD` lets the implementation decide if overallocation should be allowed.
- `VK_MEMORY_OVERALLOCATION_BEHAVIOR_ALLOWED_AMD` specifies overallocation is allowed if platform permits.
- `VK_MEMORY_OVERALLOCATION_BEHAVIOR_DISALLOWED_AMD` specifies the application is not allowed to allocate device memory beyond the heap sizes reported by `VkPhysicalDeviceMemoryProperties`. Allocations that are not explicitly made by the application within the scope of the Vulkan instance are not accounted for.

#### 4.2.2. Device Use

The following is a high-level list of `VkDevice` uses along with references on where to find more information:

- Creation of queues. See the [Queues](#) section below for further details.
- Creation and tracking of various synchronization constructs. See [Synchronization and Cache Control](#) for further details.
- Allocating, freeing, and managing memory. See [Memory Allocation](#) and [Resource Creation](#) for further details.
- Creation and destruction of command buffers and command buffer pools. See [Command Buffers](#) for further details.
- Creation, destruction, and management of graphics state. See [Pipelines](#) and [Resource Descriptors](#), among others, for further details.

#### 4.2.3. Lost Device

A logical device **may** become *lost* for a number of implementation-specific reasons, indicating that

pending and future command execution **may** fail and cause resources and backing memory to become undefined.

*Note*

Typical reasons for device loss will include things like execution timing out (to prevent denial of service), power management events, platform resource management, implementation errors.



Applications not adhering to [valid usage](#) may also result in device loss being reported, however this is not guaranteed. Even if device loss is reported, the system may be in an unrecoverable state, and further usage of the API is still considered invalid.

When this happens, certain commands will return [VK\\_ERROR\\_DEVICE\\_LOST](#) (see [Error Codes](#) for a list of such commands). After any such event, the logical device is considered *lost*. It is not possible to reset the logical device to a non-lost state, however the lost state is specific to a logical device ([VkDevice](#)), and the corresponding physical device ([VkPhysicalDevice](#)) **may** be otherwise unaffected.

In some cases, the physical device **may** also be lost, and attempting to create a new logical device will fail, returning [VK\\_ERROR\\_DEVICE\\_LOST](#). This is usually indicative of a problem with the underlying implementation, or its connection to the host. If the physical device has not been lost, and a new logical device is successfully created from that physical device, it **must** be in the non-lost state.

*Note*

Whilst logical device loss **may** be recoverable, in the case of physical device loss, it is unlikely that an application will be able to recover unless additional, unaffected physical devices exist on the system. The error is largely informational and intended only to inform the user that a platform issue has occurred, and **should** be investigated further. For example, underlying hardware **may** have developed a fault or become physically disconnected from the rest of the system. In many cases, physical device loss **may** cause other more serious issues such as the operating system crashing; in which case it **may** not be reported via the Vulkan API.



When a device is lost, its child objects are not implicitly destroyed and their handles are still valid. Those objects **must** still be destroyed before their parents or the device **can** be destroyed (see the [Object Lifetime](#) section). The host address space corresponding to device memory mapped using [vkMapMemory](#) is still valid, and host memory accesses to these mapped regions are still valid, but the contents are undefined. It is still legal to call any API command on the device and child objects.

Once a device is lost, command execution **may** fail, and commands that return a [VkResult](#) **may** return [VK\\_ERROR\\_DEVICE\\_LOST](#). Commands that do not allow run-time errors **must** still operate correctly for valid usage and, if applicable, return valid data.

Commands that wait indefinitely for device execution (namely [vkDeviceWaitIdle](#), [vkQueueWaitIdle](#), [vkWaitForFences](#) or [vkAcquireNextImageKHR](#) with a maximum [timeout](#), and [vkGetQueryPoolResults](#) with the [VK\\_QUERY\\_RESULT\\_WAIT\\_BIT](#) bit set in [flags](#)) **must** return in finite time even in the case of a lost device, and return either [VK\\_SUCCESS](#) or [VK\\_ERROR\\_DEVICE\\_LOST](#). For any

command that **may** return `VK_ERROR_DEVICE_LOST`, for the purpose of determining whether a command buffer is in the [pending state](#), or whether resources are considered in-use by the device, a return value of `VK_ERROR_DEVICE_LOST` is equivalent to `VK_SUCCESS`.

The content of any external memory objects that have been exported from or imported to a lost device become undefined. Objects on other logical devices or in other APIs which are associated with the same underlying memory resource as the external memory objects on the lost device are unaffected other than their content becoming undefined. The layout of subresources of images on other logical devices that are bound to `VkDeviceMemory` objects associated with the same underlying memory resources as external memory objects on the lost device becomes `VK_IMAGE_LAYOUT_UNDEFINED`.

The state of `VkSemaphore` objects on other logical devices created by [importing a semaphore payload](#) with temporary permanence which was exported from the lost device is undefined. The state of `VkSemaphore` objects on other logical devices that permanently share a semaphore payload with a `VkSemaphore` object on the lost device is undefined, and remains undefined following any subsequent signal operations. Implementations **must** ensure pending and subsequently submitted wait operations on such semaphores behave as defined in [Semaphore State Requirements For Wait Operations](#) for external semaphores not in a valid state for a wait operation.

#### 4.2.4. Device Destruction

To destroy a device, call:

```
void vkDestroyDevice(  
    VkDevice device,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

To ensure that no work is active on the device, `vkDeviceWaitIdle` **can** be used to gate the destruction of the device. Prior to destroying a device, an application is responsible for destroying/freeing any Vulkan objects that were created using that device as the first parameter of the corresponding `vkCreate*` or `vkAllocate*` command.

##### Note



The lifetime of each of these objects is bound by the lifetime of the `VkDevice` object. Therefore, to avoid resource leaks, it is critical that an application explicitly free all of these resources prior to calling `vkDestroyDevice`.

## Valid Usage

- All child objects created on `device` **must** have been destroyed prior to destroying `device`
- If `VkAllocationCallbacks` were provided when `device` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `device` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- If `device` is not `NULL`, `device` **must** be a valid `VkDevice` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure

## Host Synchronization

- Host access to `device` **must** be externally synchronized

## 4.3. Queues

### 4.3.1. Queue Family Properties

As discussed in the [Physical Device Enumeration](#) section above, the `vkGetPhysicalDeviceQueueFamilyProperties` command is used to retrieve details about the queue families and queues supported by a device.

Each index in the `pQueueFamilyProperties` array returned by `vkGetPhysicalDeviceQueueFamilyProperties` describes a unique queue family on that physical device. These indices are used when creating queues, and they correspond directly with the `queueFamilyIndex` that is passed to the `vkCreateDevice` command via the `VkDeviceQueueCreateInfo` structure as described in the [Queue Creation](#) section below.

Grouping of queue families within a physical device is implementation-dependent.

#### Note



The general expectation is that a physical device groups all queues of matching capabilities into a single family. However, while implementations **should** do this, it is possible that a physical device **may** return two separate queue families with the same capabilities.

Once an application has identified a physical device with the queue(s) that it desires to use, it will create those queues in conjunction with a logical device. This is described in the following section.

### 4.3.2. Queue Creation

Creating a logical device also creates the queues associated with that device. The queues to create are described by a set of [VkDeviceQueueCreateInfo](#) structures that are passed to [vkCreateDevice](#) in `pQueueCreateInfos`.

Queues are represented by [VkQueue](#) handles:

```
VK_DEFINE_HANDLE(VkQueue)
```

The [VkDeviceQueueCreateInfo](#) structure is defined as:

```
typedef struct VkDeviceQueueCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkDeviceQueueCreateFlags  flags;
    uint32_t                 queueFamilyIndex;
    uint32_t                 queueCount;
    const float*             pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask indicating behavior of the queue.
- `queueFamilyIndex` is an unsigned integer indicating the index of the queue family to create on this device. This index corresponds to the index of an element of the `pQueueFamilyProperties` array that was returned by [vkGetPhysicalDeviceQueueFamilyProperties](#).
- `queueCount` is an unsigned integer specifying the number of queues to create in the queue family indicated by `queueFamilyIndex`.
- `pQueuePriorities` is a pointer to an array of `queueCount` normalized floating point values, specifying priorities of work that will be submitted to each created queue. See [Queue Priority](#) for more information.

### Valid Usage

- `queueFamilyIndex` **must** be less than `pQueueFamilyPropertyCount` returned by [vkGetPhysicalDeviceQueueFamilyProperties](#)
- `queueCount` **must** be less than or equal to the `queueCount` member of the `VkQueueFamilyProperties` structure, as returned by [vkGetPhysicalDeviceQueueFamilyProperties](#) in the `pQueueFamilyProperties[queueFamilyIndex]`
- Each element of `pQueuePriorities` **must** be between `0.0` and `1.0` inclusive

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO`
- **pNext** must be `NULL` or a pointer to a valid instance of `VkDeviceQueueGlobalPriorityCreateInfoEXT`
- **flags** must be a valid combination of `VkDeviceQueueCreateFlagBits` values
- **pQueuePriorities** must be a valid pointer to an array of `queueCount float` values
- **queueCount** must be greater than `0`

Bits which **can** be set in `VkDeviceQueueCreateInfo::flags` to specify usage behavior of the queue are:

```
typedef enum VkDeviceQueueCreateFlagBits {
    VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT = 0x00000001,
    VK_DEVICE_QUEUE_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkDeviceQueueCreateFlagBits;
```

- `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT` specifies that the device queue is a protected-capable queue. If the protected memory feature is not enabled, the `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT` bit of `flags` must not be set.

```
typedef VkFlags VkDeviceQueueCreateFlags;
```

`VkDeviceQueueCreateFlags` is a bitmask type for setting a mask of zero or more `VkDeviceQueueCreateFlagBits`.

A queue can be created with a system-wide priority by including a `VkDeviceQueueGlobalPriorityCreateInfoEXT` structure in the `pNext` chain of `VkDeviceQueueCreateInfo`.

The `VkDeviceQueueGlobalPriorityCreateInfoEXT` structure is defined as:

```
typedef struct VkDeviceQueueGlobalPriorityCreateInfoEXT {
    VkStructureType           sType;
    const void*               pNext;
    VkQueueGlobalPriorityEXT  globalPriority;
} VkDeviceQueueGlobalPriorityCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `globalPriority` is the system-wide priority associated to this queue as specified by `VkQueueGlobalPriorityEXT`

A queue created without specifying `VkDeviceQueueGlobalPriorityCreateInfoEXT` will default to `VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT`
- `globalPriority` must be a valid `VkQueueGlobalPriorityEXT` value

Possible values of `VkDeviceQueueGlobalPriorityCreateInfoEXT::globalPriority`, specifying a system-wide priority level are:

```
typedef enum VkQueueGlobalPriorityEXT {
    VK_QUEUE_GLOBAL_PRIORITY_LOW_EXT = 128,
    VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT = 256,
    VK_QUEUE_GLOBAL_PRIORITY_HIGH_EXT = 512,
    VK_QUEUE_GLOBAL_PRIORITY_REALTIME_EXT = 1024,
    VK_QUEUE_GLOBAL_PRIORITY_MAX_ENUM_EXT = 0x7FFFFFFF
} VkQueueGlobalPriorityEXT;
```

Priority values are sorted in ascending order. A comparison operation on the enum values can be used to determine the priority order.

- `VK_QUEUE_GLOBAL_PRIORITY_LOW_EXT` is below the system default. Useful for non-interactive tasks.
- `VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT` is the system default priority.
- `VK_QUEUE_GLOBAL_PRIORITY_HIGH_EXT` is above the system default.
- `VK_QUEUE_GLOBAL_PRIORITY_REALTIME_EXT` is the highest priority. Useful for critical tasks.

Queues with higher system priority **may** be allotted more processing time than queues with lower priority. An implementation **may** allow a higher-priority queue to starve a lower-priority queue until the higher-priority queue has no further commands to execute.

Priorities imply no ordering or scheduling constraints.

No specific guarantees are made about higher priority queues receiving more processing time or better quality of service than lower priority queues.

The global priority level of a queue takes precedence over the per-process queue priority (`VkDeviceQueueCreateInfo::pQueuePriorities`).

Abuse of this feature **may** result in starving the rest of the system of implementation resources. Therefore, the driver implementation **may** deny requests to acquire a priority above the default priority (`VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT`) if the caller does not have sufficient privileges. In this scenario `VK_ERROR_NOT_PERMITTED_EXT` is returned.

The driver implementation **may** fail the queue allocation request if resources required to complete the operation have been exhausted (either by the same process or a different process). In this

scenario `VK_ERROR_INITIALIZATION_FAILED` is returned.

To retrieve a handle to a `VkQueue` object, call:

```
void vkGetDeviceQueue(  
    VkDevice  
    uint32_t  
    uint32_t  
    VkQueue*  
                                device,  
                                queueFamilyIndex,  
                                queueIndex,  
                                pQueue);
```

- `device` is the logical device that owns the queue.
- `queueFamilyIndex` is the index of the queue family to which the queue belongs.
- `queueIndex` is the index within this queue family of the queue to retrieve.
- `pQueue` is a pointer to a `VkQueue` object that will be filled with the handle for the requested queue.

`vkGetDeviceQueue` **must** only be used to get queues that were created with the `flags` parameter of `VkDeviceQueueCreateInfo` set to zero. To get queues that were created with a non-zero `flags` parameter use `vkGetDeviceQueue2`.

## Valid Usage

- `queueFamilyIndex` **must** be one of the queue family indices specified when `device` was created, via the `VkDeviceQueueCreateInfo` structure
- `queueIndex` **must** be less than the number of queues created for the specified queue family index when `device` was created, via the `queueCount` member of the `VkDeviceQueueCreateInfo` structure
- `VkDeviceQueueCreateInfo::flags` **must** have been set to zero when `device` was created

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pQueue` **must** be a valid pointer to a `VkQueue` handle

To retrieve a handle to a `VkQueue` object with specific `VkDeviceQueueCreateFlags` creation flags, call:

```
void vkGetDeviceQueue2(  
    VkDevice  
    const VkDeviceQueueCreateInfo*  
    VkQueue*  
                                device,  
                                pCreateInfo,  
                                pQueue);
```

- `device` is the logical device that owns the queue.

- `pQueueInfo` is a pointer to a `VkDeviceCreateInfo2` structure, describing the parameters used to create the device queue.
- `pQueue` is a pointer to a `VkQueue` object that will be filled with the handle for the requested queue.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pQueueInfo` **must** be a valid pointer to a valid `VkDeviceCreateInfo2` structure
- `pQueue` **must** be a valid pointer to a `VkQueue` handle

The `VkDeviceCreateInfo2` structure is defined as:

```
typedef struct VkDeviceCreateInfo2 {
    VkStructureType           sType;
    const void*               pNext;
    VkDeviceQueueCreateFlags   flags;
    uint32_t                  queueFamilyIndex;
    uint32_t                  queueIndex;
} VkDeviceCreateInfo2;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure. The `pNext` chain of `VkDeviceCreateInfo2` is used to provide additional image parameters to `vkGetDeviceQueue2`.
- `flags` is a `VkDeviceQueueCreateFlags` value indicating the flags used to create the device queue.
- `queueFamilyIndex` is the index of the queue family to which the queue belongs.
- `queueIndex` is the index within this queue family of the queue to retrieve.

The queue returned by `vkGetDeviceQueue2` **must** have the same `flags` value from this structure as that used at device creation time in a `VkDeviceQueueCreateInfo` instance. If no matching `flags` were specified at device creation time then `pQueue` will return `VK_NULL_HANDLE`.

## Valid Usage

- `queueFamilyIndex` **must** be one of the queue family indices specified when `device` was created, via the `VkDeviceCreateInfo` structure
- `queueIndex` **must** be less than the number of queues created for the specified queue family index and `VkDeviceQueueCreateInfo` member `flags` equal to this `flags` value when `device` was created, via the `queueCount` member of the `VkDeviceCreateInfo` structure

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of `VkDeviceQueueCreateInfoFlagBits` values
- `flags` **must** not be `0`

### 4.3.3. Queue Family Index

The queue family index is used in multiple places in Vulkan in order to tie operations to a specific family of queues.

When retrieving a handle to the queue via `vkGetDeviceQueue`, the queue family index is used to select which queue family to retrieve the `VkQueue` handle from as described in the previous section.

When creating a `VkCommandPool` object (see [Command Pools](#)), a queue family index is specified in the `VkCommandPoolCreateInfo` structure. Command buffers from this pool **can** only be submitted on queues corresponding to this queue family.

When creating `VkImage` (see [Images](#)) and `VkBuffer` (see [Buffers](#)) resources, a set of queue families is included in the `VkImageCreateInfo` and `VkBufferCreateInfo` structures to specify the queue families that **can** access the resource.

When inserting a `VkBufferMemoryBarrier` or `VkImageMemoryBarrier` (see [Pipeline Barriers](#)), a source and destination queue family index is specified to allow the ownership of a buffer or image to be transferred from one queue family to another. See the [Resource Sharing](#) section for details.

### 4.3.4. Queue Priority

Each queue is assigned a priority, as set in the `VkDeviceQueueCreateInfo` structures when creating the device. The priority of each queue is a normalized floating point value between 0.0 and 1.0, which is then translated to a discrete priority level by the implementation. Higher values indicate a higher priority, with 0.0 being the lowest priority and 1.0 being the highest.

Within the same device, queues with higher priority **may** be allotted more processing time than queues with lower priority. The implementation makes no guarantees with regards to ordering or scheduling among queues with the same priority, other than the constraints defined by any [explicit synchronization primitives](#). The implementation make no guarantees with regards to queues across different devices.

An implementation **may** allow a higher-priority queue to starve a lower-priority queue on the same `VkDevice` until the higher-priority queue has no further commands to execute. The relationship of queue priorities **must** not cause queues on one `VkDevice` to starve queues on another `VkDevice`.

No specific guarantees are made about higher priority queues receiving more processing time or better quality of service than lower priority queues.

### 4.3.5. Queue Submission

Work is submitted to a queue via *queue submission* commands such as [vkQueueSubmit](#). Queue submission commands define a set of *queue operations* to be executed by the underlying physical device, including synchronization with semaphores and fences.

Submission commands take as parameters a target queue, zero or more *batches* of work, and an **optional** fence to signal upon completion. Each batch consists of three distinct parts:

1. Zero or more semaphores to wait on before execution of the rest of the batch.
  - If present, these describe a [semaphore wait operation](#).
2. Zero or more work items to execute.
  - If present, these describe a *queue operation* matching the work described.
3. Zero or more semaphores to signal upon completion of the work items.
  - If present, these describe a [semaphore signal operation](#).

If a fence is present in a queue submission, it describes a [fence signal operation](#).

All work described by a queue submission command **must** be submitted to the queue before the command returns.

### Sparse Memory Binding

In Vulkan it is possible to sparsely bind memory to buffers and images as described in the [Sparse Resource](#) chapter. Sparse memory binding is a queue operation. A queue whose flags include the `VK_QUEUE_SPARSE_BINDING_BIT` **must** be able to support the mapping of a virtual address to a physical address on the device. This causes an update to the page table mappings on the device. This update **must** be synchronized on a queue to avoid corrupting page table mappings during execution of graphics commands. By binding the sparse memory resources on queues, all commands that are dependent on the updated bindings are synchronized to only execute after the binding is updated. See the [Synchronization and Cache Control](#) chapter for how this synchronization is accomplished.

### 4.3.6. Queue Destruction

Queues are created along with a logical device during [vkCreateDevice](#). All queues associated with a logical device are destroyed when [vkDestroyDevice](#) is called on that device.

# Chapter 5. Command Buffers

Command buffers are objects used to record commands which **can** be subsequently submitted to a device queue for execution. There are two levels of command buffers - *primary command buffers*, which **can** execute secondary command buffers, and which are submitted to queues, and *secondary command buffers*, which **can** be executed by primary command buffers, and which are not directly submitted to queues.

Command buffers are represented by `VkCommandBuffer` handles:

```
VK_DEFINE_HANDLE(VkCommandBuffer)
```

Recorded commands include commands to bind pipelines and descriptor sets to the command buffer, commands to modify dynamic state, commands to draw (for graphics rendering), commands to dispatch (for compute), commands to execute secondary command buffers (for primary command buffers only), commands to copy buffers and images, and other commands.

Each command buffer manages state independently of other command buffers. There is no inheritance of state across primary and secondary command buffers, or between secondary command buffers. When a command buffer begins recording, all state in that command buffer is undefined. When secondary command buffer(s) are recorded to execute on a primary command buffer, the secondary command buffer inherits no state from the primary command buffer, and all state of the primary command buffer is undefined after an execute secondary command buffer command is recorded. There is one exception to this rule - if the primary command buffer is inside a render pass instance, then the render pass and subpass state is not disturbed by executing secondary command buffers. For state dependent commands (such as draws and dispatches), any state consumed by those commands **must** not be undefined.

Unless otherwise specified, and without explicit synchronization, the various commands submitted to a queue via command buffers **may** execute in arbitrary order relative to each other, and/or concurrently. Also, the memory side-effects of those commands **may** not be directly visible to other commands without explicit memory dependencies. This is true within a command buffer, and across command buffers submitted to a given queue. See [the synchronization chapter](#) for information on [implicit](#) and explicit synchronization between commands.

## 5.1. Command Buffer Lifecycle

Each command buffer is always in one of the following states:

### Initial

When a command buffer is [allocated](#), it is in the *initial state*. Some commands are able to *reset* a command buffer, or a set of command buffers, back to this state from any of the executable, recording or invalid state. Command buffers in the initial state **can** only be moved to the recording state, or freed.

### Recording

[vkBeginCommandBuffer](#) changes the state of a command buffer from the initial state to the

*recording state*. Once a command buffer is in the recording state, `vkCmd*` commands **can** be used to record to the command buffer.

## Executable

`vkEndCommandBuffer` ends the recording of a command buffer, and moves it from the recording state to the *executable state*. Executable command buffers **can** be [submitted](#), reset, or [recorded to another command buffer](#).

## Pending

[Queue submission](#) of a command buffer changes the state of a command buffer from the executable state to the *pending state*. Whilst in the pending state, applications **must** not attempt to modify the command buffer in any way - as the device **may** be processing the commands recorded to it. Once execution of a command buffer completes, the command buffer reverts back to either the *executable state*, or the *invalid state* if it was recorded with `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`. A [synchronization](#) command **should** be used to detect when this occurs.

## Invalid

Some operations, such as [modifying or deleting a resource](#) that was used in a command recorded to a command buffer, will transition the state of that command buffer into the *invalid state*. Command buffers in the invalid state **can** only be reset or freed.

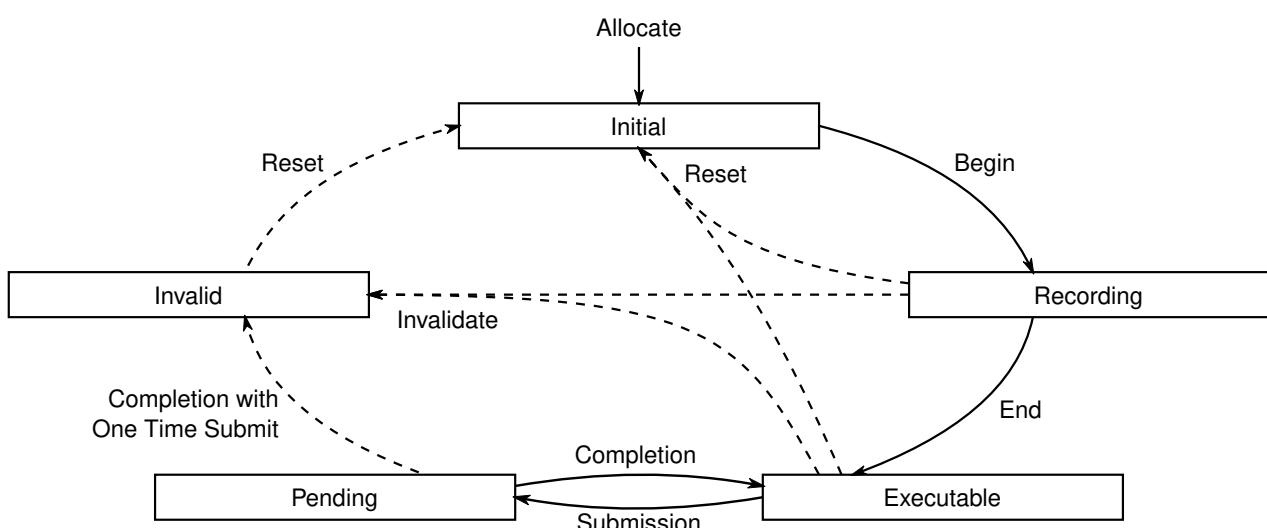


Figure 1. Lifecycle of a command buffer

Any given command that operates on a command buffer has its own requirements on what state a command buffer **must** be in, which are detailed in the valid usage constraints for that command.

Resetting a command buffer is an operation that discards any previously recorded commands and puts a command buffer in the initial state. Resetting occurs as a result of `vkResetCommandBuffer` or `vkResetCommandPool`, or as part of `vkBeginCommandBuffer` (which additionally puts the command buffer in the recording state).

Secondary command buffers **can** be recorded to a primary command buffer via `vkCmdExecuteCommands`. This partially ties the lifecycle of the two command buffers together - if the primary is submitted to a queue, both the primary and any secondaries recorded to it move to the pending state. Once execution of the primary completes, so does any secondary recorded within

it, and once all executions of each command buffer complete, they move to the executable state. If a secondary moves to any other state whilst it is recorded to another command buffer, the primary moves to the invalid state. A primary moving to any other state does not affect the state of the secondary. Resetting or freeing a primary command buffer removes the linkage to any secondary command buffers that were recorded to it.

## 5.2. Command Pools

Command pools are opaque objects that command buffer memory is allocated from, and which allow the implementation to amortize the cost of resource creation across multiple command buffers. Command pools are externally synchronized, meaning that a command pool **must** not be used concurrently in multiple threads. That includes use via recording commands on any command buffers allocated from the pool, as well as operations that allocate, free, and reset command buffers or the pool itself.

Command pools are represented by `VkCommandPool` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkCommandPool)
```

To create a command pool, call:

```
VkResult vkCreateCommandPool(  
    VkDevice                                     device,  
    const VkCommandPoolCreateInfo*                pCreateInfo,  
    const VkAllocationCallbacks*                  pAllocator,  
    VkCommandPool*                                pCommandPool);
```

- `device` is the logical device that creates the command pool.
- `pCreateInfo` is a pointer to a `VkCommandPoolCreateInfo` structure specifying the state of the command pool object.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pCommandPool` is a pointer to a `VkCommandPool` handle in which the created pool is returned.

### Valid Usage

- `pCreateInfo::queueFamilyIndex` **must** be the index of a queue family available in the logical device `device`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkCommandPoolCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pCommandPool` **must** be a valid pointer to a `VkCommandPool` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkCommandPoolCreateInfo` structure is defined as:

```
typedef struct VkCommandPoolCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkCommandPoolCreateFlags   flags;
    uint32_t                  queueFamilyIndex;
} VkCommandPoolCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkCommandPoolCreateFlagBits` indicating usage behavior for the pool and command buffers allocated from it.
- `queueFamilyIndex` designates a queue family as described in section [Queue Family Properties](#). All command buffers allocated from this command pool **must** be submitted on queues from the same queue family.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of `VkCommandPoolCreateFlagBits` values

Bits which **can** be set in `VkCommandPoolCreateInfo::flags` to specify usage behavior for a

command pool are:

```
typedef enum VkCommandPoolCreateFlagBits {
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT = 0x00000001,
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT = 0x00000002,
    VK_COMMAND_POOL_CREATE_PROTECTED_BIT = 0x00000004,
    VK_COMMAND_POOL_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkCommandPoolCreateFlagBits;
```

- `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` specifies that command buffers allocated from the pool will be short-lived, meaning that they will be reset or freed in a relatively short timeframe. This flag **may** be used by the implementation to control memory allocation behavior within the pool.
- `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` allows any command buffer allocated from a pool to be individually reset to the [initial state](#); either by calling `vkResetCommandBuffer`, or via the implicit reset when calling `vkBeginCommandBuffer`. If this flag is not set on a pool, then `vkResetCommandBuffer` **must** not be called for any command buffer allocated from that pool.
- `VK_COMMAND_POOL_CREATE_PROTECTED_BIT` specifies that command buffers allocated from the pool are protected command buffers. If the protected memory feature is not enabled, the `VK_COMMAND_POOL_CREATE_PROTECTED_BIT` bit of `flags` **must** not be set.

```
typedef VkFlags VkCommandPoolCreateFlags;
```

`VkCommandPoolCreateFlags` is a bitmask type for setting a mask of zero or more `VkCommandPoolCreateFlagBits`.

To trim a command pool, call:

```
void vkTrimCommandPool(
    VkDevice                               device,
    VkCommandPool                         commandPool,
    VkCommandPoolTrimFlags                flags);
```

or the equivalent command

```
void vkTrimCommandPoolKHR(
    VkDevice                               device,
    VkCommandPool                         commandPool,
    VkCommandPoolTrimFlags                flags);
```

- `device` is the logical device that owns the command pool.
- `commandPool` is the command pool to trim.
- `flags` is reserved for future use.

Trimming a command pool recycles unused memory from the command pool back to the system.

Command buffers allocated from the pool are not affected by the command.

*Note*

This command provides applications with some control over the internal memory allocations used by command pools.

Unused memory normally arises from command buffers that have been recorded and later reset, such that they are no longer using the memory. On reset, a command buffer can return memory to its command pool, but the only way to release memory from a command pool to the system requires calling [vkResetCommandPool](#), which cannot be executed while any command buffers from that pool are still in use. Subsequent recording operations into command buffers will re-use this memory but since total memory requirements fluctuate over time, unused memory can accumulate.

In this situation, trimming a command pool **may** be useful to return unused memory back to the system, returning the total outstanding memory allocated by the pool back to a more “average” value.



Implementations utilize many internal allocation strategies that make it impossible to guarantee that all unused memory is released back to the system. For instance, an implementation of a command pool **may** involve allocating memory in bulk from the system and sub-allocating from that memory. In such an implementation any live command buffer that holds a reference to a bulk allocation would prevent that allocation from being freed, even if only a small proportion of the bulk allocation is in use.

In most cases trimming will result in a reduction in allocated but unused memory, but it does not guarantee the “ideal” behavior.

Trimming **may** be an expensive operation, and **should** not be called frequently. Trimming **should** be treated as a way to relieve memory pressure after application-known points when there exists enough unused memory that the cost of trimming is “worth” it.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `commandPool` **must** be a valid `VkCommandPool` handle
- `flags` **must** be `0`
- `commandPool` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `commandPool` **must** be externally synchronized

```
typedef VkFlags VkCommandPoolTrimFlags;
```

or the equivalent

```
typedef VkCommandPoolTrimFlags VkCommandPoolTrimFlagsKHR;
```

`VkCommandPoolTrimFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To reset a command pool, call:

```
VkResult vkResetCommandPool(  
    VkDevice device,  
    VkCommandPool commandPool,  
    VkCommandPoolResetFlags flags);
```

- `device` is the logical device that owns the command pool.
- `commandPool` is the command pool to reset.
- `flags` is a bitmask of `VkCommandPoolResetFlagBits` controlling the reset operation.

Resetting a command pool recycles all of the resources from all of the command buffers allocated from the command pool back to the command pool. All command buffers that have been allocated from the command pool are put in the [initial state](#).

Any primary command buffer allocated from another `VkCommandPool` that is in the [recording or executable state](#) and has a secondary command buffer allocated from `commandPool` recorded into it, becomes [invalid](#).

## Valid Usage

- All `VkCommandBuffer` objects allocated from `commandPool` **must** not be in the [pending state](#)

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `commandPool` **must** be a valid `VkCommandPool` handle
- `flags` **must** be a valid combination of `VkCommandPoolResetFlagBits` values
- `commandPool` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `commandPool` must be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Bits which **can** be set in `vkResetCommandPool::flags` to control the reset operation are:

```
typedef enum VkCommandPoolResetFlagBits {
    VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
    VK_COMMAND_POOL_RESET_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkCommandPoolResetFlagBits;
```

- `VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT` specifies that resetting a command pool recycles all of the resources from the command pool back to the system.

```
typedef VkFlags VkCommandPoolResetFlags;
```

`VkCommandPoolResetFlags` is a bitmask type for setting a mask of zero or more `VkCommandPoolResetFlagBits`.

To destroy a command pool, call:

```
void vkDestroyCommandPool(
    VkDevice                                     device,
    VkCommandPool                                commandPool,
    const VkAllocationCallbacks*                  pAllocator);
```

- `device` is the logical device that destroys the command pool.
- `commandPool` is the handle of the command pool to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

When a pool is destroyed, all command buffers allocated from the pool are [freed](#).

Any primary command buffer allocated from another `VkCommandPool` that is in the [recording](#) or [executable state](#) and has a secondary command buffer allocated from `commandPool` recorded into it,

becomes [invalid](#).

## Valid Usage

- All `VkCommandBuffer` objects allocated from `commandPool` **must** not be in the [pending state](#).
- If `VkAllocationCallbacks` were provided when `commandPool` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `commandPool` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `commandPool` is not `VK_NULL_HANDLE`, `commandPool` **must** be a valid `VkCommandPool` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `commandPool` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `commandPool` **must** be externally synchronized

## 5.3. Command Buffer Allocation and Management

To allocate command buffers, call:

```
VkResult vkAllocateCommandBuffers(
    VkDevice                                     device,
    const VkCommandBufferAllocateInfo* pAllocateInfo,
    VkCommandBuffer*                            pCommandBuffers);
```

- `device` is the logical device that owns the command pool.
- `pAllocateInfo` is a pointer to a `VkCommandBufferAllocateInfo` structure describing parameters of the allocation.
- `pCommandBuffers` is a pointer to an array of `VkCommandBuffer` handles in which the resulting command buffer objects are returned. The array **must** be at least the length specified by the `commandBufferCount` member of `pAllocateInfo`. Each allocated command buffer begins in the initial state.

`vkAllocateCommandBuffers` **can** be used to create multiple command buffers. If the creation of any of

those command buffers fails, the implementation **must** destroy all successfully created command buffer objects from this command, set all entries of the `pCommandBuffers` array to `NULL` and return the error.

When command buffers are first allocated, they are in the [initial state](#).

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pAllocateInfo` **must** be a valid pointer to a valid `VkCommandBufferAllocateInfo` structure
- `pCommandBuffers` **must** be a valid pointer to an array of `pAllocateInfo::commandBufferCount` `VkCommandBuffer` handles
- The value referenced by `pAllocateInfo::commandBufferCount` **must** be greater than `0`

## Host Synchronization

- Host access to `pAllocateInfo::commandPool` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkCommandBufferAllocateInfo` structure is defined as:

```
typedef struct VkCommandBufferAllocateInfo {
    VkStructureType      sType;
    const void*          pNext;
    VkCommandPool        commandPool;
    VkCommandBufferLevel level;
    uint32_t             commandBufferCount;
} VkCommandBufferAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `commandPool` is the command pool from which the command buffers are allocated.
- `level` is a `VkCommandBufferLevel` value specifying the command buffer level.
- `commandBufferCount` is the number of command buffers to allocate from the pool.

## Valid Usage

- `commandBufferCount` **must** be greater than `0`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO`
- `pNext` **must** be `NULL`
- `commandPool` **must** be a valid `VkCommandPool` handle
- `level` **must** be a valid `VkCommandBufferLevel` value

Possible values of `VkCommandBufferAllocateInfo::level`, specifying the command buffer level, are:

```
typedef enum VkCommandBufferLevel {  
    VK_COMMAND_BUFFER_LEVEL_PRIMARY = 0,  
    VK_COMMAND_BUFFER_LEVEL_SECONDARY = 1,  
    VK_COMMAND_BUFFER_LEVEL_MAX_ENUM = 0x7FFFFFFF  
} VkCommandBufferLevel;
```

- `VK_COMMAND_BUFFER_LEVEL_PRIMARY` specifies a primary command buffer.
- `VK_COMMAND_BUFFER_LEVEL_SECONDARY` specifies a secondary command buffer.

To reset command buffers, call:

```
VkResult vkResetCommandBuffer(  
    VkCommandBuffer  
    VkCommandBufferResetFlags  
        commandBuffer,  
        flags);
```

- `commandBuffer` is the command buffer to reset. The command buffer **can** be in any state other than `pending`, and is moved into the `initial state`.
- `flags` is a bitmask of `VkCommandBufferResetFlagBits` controlling the reset operation.

Any primary command buffer that is in the `recording or executable state` and has `commandBuffer` recorded into it, becomes `invalid`.

## Valid Usage

- `commandBuffer` **must** not be in the `pending state`
- `commandBuffer` **must** have been allocated from a pool that was created with the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `flags` **must** be a valid combination of `VkCommandBufferResetFlagBits` values

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Bits which **can** be set in `vkResetCommandBuffer::flags` to control the reset operation are:

```
typedef enum VkCommandBufferResetFlagBits {
    VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
    VK_COMMAND_BUFFER_RESET_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkCommandBufferResetFlagBits;
```

- `VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT` specifies that most or all memory resources currently owned by the command buffer **should** be returned to the parent command pool. If this flag is not set, then the command buffer **may** hold onto memory resources and reuse them when recording commands. `commandBuffer` is moved to the [initial state](#).

```
typedef VkFlags VkCommandBufferResetFlags;
```

`VkCommandBufferResetFlags` is a bitmask type for setting a mask of zero or more `VkCommandBufferResetFlagBits`.

To free command buffers, call:

```
void vkFreeCommandBuffers(
    VkDevice                                     device,
    VkCommandPool                                commandPool,
    uint32_t                                      commandBufferCount,
    const VkCommandBuffer*                        pCommandBuffers);
```

- `device` is the logical device that owns the command pool.
- `commandPool` is the command pool from which the command buffers were allocated.
- `commandBufferCount` is the length of the `pCommandBuffers` array.
- `pCommandBuffers` is a pointer to an array of handles of command buffers to free.

Any primary command buffer that is in the [recording or executable state](#) and has any element of `pCommandBuffers` recorded into it, becomes [invalid](#).

### Valid Usage

- All elements of `pCommandBuffers` **must** not be in the [pending state](#)
- `pCommandBuffers` **must** be a valid pointer to an array of `commandBufferCount` `VkCommandBuffer` handles, each element of which **must** either be a valid handle or `NULL`

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `commandPool` **must** be a valid `VkCommandPool` handle
- `commandBufferCount` **must** be greater than `0`
- `commandPool` **must** have been created, allocated, or retrieved from `device`
- Each element of `pCommandBuffers` that is a valid handle **must** have been created, allocated, or retrieved from `commandPool`

### Host Synchronization

- Host access to `commandPool` **must** be externally synchronized
- Host access to each member of `pCommandBuffers` **must** be externally synchronized

## 5.4. Command Buffer Recording

To begin recording a command buffer, call:

```
VkResult vkBeginCommandBuffer(
    VkCommandBuffer
    const VkCommandBufferBeginInfo*           commandBuffer,
                                            pBeginInfo);
```

- `commandBuffer` is the handle of the command buffer which is to be put in the recording state.
- `pBeginInfo` points to a `VkCommandBufferBeginInfo` structure defining additional information about how the command buffer begins recording.

## Valid Usage

- `commandBuffer` **must** not be in the `recording` or `pending` state.
- If `commandBuffer` was allocated from a `VkCommandPool` which did not have the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` flag set, `commandBuffer` **must** be in the `initial` state.
- If `commandBuffer` is a secondary command buffer, the `pInheritanceInfo` member of `pBeginInfo` **must** be a valid `VkCommandBufferInheritanceInfo` structure
- If `commandBuffer` is a secondary command buffer and either the `occlusionQueryEnable` member of the `pInheritanceInfo` member of `pBeginInfo` is `VK_FALSE`, or the precise occlusion queries feature is not enabled, the `queryFlags` member of the `pInheritanceInfo` member `pBeginInfo` **must** not contain `VK_QUERY_CONTROL_PRECISE_BIT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pBeginInfo` **must** be a valid pointer to a valid `VkCommandBufferBeginInfo` structure

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkCommandBufferBeginInfo` structure is defined as:

```
typedef struct VkCommandBufferBeginInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkCommandBufferUsageFlags flags;
    const VkCommandBufferInheritanceInfo* pInheritanceInfo;
} VkCommandBufferBeginInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkCommandBufferUsageFlagBits` specifying usage behavior for the command buffer.
- `pInheritanceInfo` is a pointer to a `VkCommandBufferInheritanceInfo` structure, used if `commandBuffer` is a secondary command buffer. If this is a primary command buffer, then this value is ignored.

## Valid Usage

- If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `renderPass` member of `pInheritanceInfo` **must** be a valid `VkRenderPass`
- If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `subpass` member of `pInheritanceInfo` **must** be a valid subpass index within the `renderPass` member of `pInheritanceInfo`
- If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `framebuffer` member of `pInheritanceInfo` **must** be either `VK_NULL_HANDLE`, or a valid `VkFramebuffer` that is compatible with the `renderPass` member of `pInheritanceInfo`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkDeviceGroupCommandBufferBeginInfo`
- `flags` **must** be a valid combination of `VkCommandBufferUsageFlagBits` values

Bits which **can** be set in `VkCommandBufferBeginInfo::flags` to specify usage behavior for a command buffer are:

```
typedef enum VkCommandBufferUsageFlagBits {
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT = 0x00000001,
    VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT = 0x00000002,
    VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT = 0x00000004,
    VK_COMMAND_BUFFER_USAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkCommandBufferUsageFlagBits;
```

- `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` specifies that each recording of the command buffer will only be submitted once, and the command buffer will be reset and recorded again between each submission.
- `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` specifies that a secondary command buffer is considered to be entirely inside a render pass. If this is a primary command buffer, then this bit is ignored.

- `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` specifies that a command buffer **can** be resubmitted to a queue while it is in the *pending state*, and recorded into multiple primary command buffers.

```
typedef VkFlags VkCommandBufferUsageFlags;
```

`VkCommandBufferUsageFlags` is a bitmask type for setting a mask of zero or more `VkCommandBufferUsageFlagBits`.

If the command buffer is a secondary command buffer, then the `VkCommandBufferInheritanceInfo` structure defines any state that will be inherited from the primary command buffer:

```
typedef struct VkCommandBufferInheritanceInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkRenderPass              renderPass;
    uint32_t                  subpass;
    VkFramebuffer              framebuffer;
    VkBool32                  occlusionQueryEnable;
    VkQueryControlFlags       queryFlags;
    VkQueryPipelineStatisticFlags pipelineStatistics;
} VkCommandBufferInheritanceInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `renderPass` is a `VkRenderPass` object defining which render passes the `VkCommandBuffer` will be **compatible** with and **can** be executed within. If the `VkCommandBuffer` will not be executed within a render pass instance, `renderPass` is ignored.
- `subpass` is the index of the subpass within the render pass instance that the `VkCommandBuffer` will be executed within. If the `VkCommandBuffer` will not be executed within a render pass instance, `subpass` is ignored.
- `framebuffer` optionally refers to the `VkFramebuffer` object that the `VkCommandBuffer` will be rendering to if it is executed within a render pass instance. It **can** be `VK_NULL_HANDLE` if the framebuffer is not known, or if the `VkCommandBuffer` will not be executed within a render pass instance.

#### Note



Specifying the exact framebuffer that the secondary command buffer will be executed with **may** result in better performance at command buffer execution time.

- `occlusionQueryEnable` specifies whether the command buffer **can** be executed while an occlusion query is active in the primary command buffer. If this is `VK_TRUE`, then this command buffer **can** be executed whether the primary command buffer has an occlusion query active or not. If this is `VK_FALSE`, then the primary command buffer **must** not have an occlusion query

active.

- `queryFlags` specifies the query flags that **can** be used by an active occlusion query in the primary command buffer when this secondary command buffer is executed. If this value includes the `VK_QUERY_CONTROL_PRECISE_BIT` bit, then the active query **can** return boolean results or actual sample counts. If this bit is not set, then the active query **must** not use the `VK_QUERY_CONTROL_PRECISE_BIT` bit.
- `pipelineStatistics` is a bitmask of `VkQueryPipelineStatisticFlagBits` specifying the set of pipeline statistics that **can** be counted by an active query in the primary command buffer when this secondary command buffer is executed. If this value includes a given bit, then this command buffer **can** be executed whether the primary command buffer has a pipeline statistics query active that includes this bit or not. If this value excludes a given bit, then the active pipeline statistics query **must** not be from a query pool that counts that statistic.

## Valid Usage

- If the `inherited queries` feature is not enabled, `occlusionQueryEnable` **must** be `VK_FALSE`
- If the `inherited queries` feature is enabled, `queryFlags` **must** be a valid combination of `VkQueryControlFlagBits` values
- If the `inherited queries` feature is not enabled, `queryFlags` **must** be `0`
- If the `pipeline statistics queries` feature is enabled, `pipelineStatistics` **must** be a valid combination of `VkQueryPipelineStatisticFlagBits` values
- If the `pipeline statistics queries` feature is not enabled, `pipelineStatistics` **must** be `0`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkCommandBufferInheritanceConditionalRenderingInfoEXT`
- Both of `framebuffer`, and `renderPass` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

If `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` was not set when creating a command buffer, that command buffer **must** not be submitted to a queue whilst it is already in the `pending state`. If `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` is not set on a secondary command buffer, that command buffer **must** not be used more than once in a given primary command buffer.

### Note



On some implementations, not using the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` bit enables command buffers to be patched in-place if needed, rather than creating a copy of the command buffer.

If a command buffer is in the `invalid, or executable state`, and the command buffer was allocated

from a command pool with the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` flag set, then `vkBeginCommandBuffer` implicitly resets the command buffer, behaving as if `vkResetCommandBuffer` had been called with `VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT` not set. After the implicit reset, `commandBuffer` is moved to the [recording state](#).

If the `pNext` chain of `VkCommandBufferInheritanceInfo` includes a `VkCommandBufferInheritanceConditionalRenderingInfoEXT` structure, then that structure controls whether a command buffer **can** be executed while conditional rendering is [active](#) in the primary command buffer.

The `VkCommandBufferInheritanceConditionalRenderingInfoEXT` structure is defined as:

```
typedef struct VkCommandBufferInheritanceConditionalRenderingInfoEXT {
    VkStructureType sType;
    const void*     pNext;
    VkBool32        conditionalRenderingEnable;
} VkCommandBufferInheritanceConditionalRenderingInfoEXT;
```

- `sType` is the type of this structure
- `pNext` is `NULL` or a pointer to an extension-specific structure
- `conditionalRenderingEnable` specifies whether the command buffer **can** be executed while conditional rendering is active in the primary command buffer. If this is `VK_TRUE`, then this command buffer **can** be executed whether the primary command buffer has active conditional rendering or not. If this is `VK_FALSE`, then the primary command buffer **must** not have conditional rendering active.

If this structure is not present, the behavior is as if `conditionalRenderingEnable` is `VK_FALSE`.

## Valid Usage

- If the [inherited conditional rendering](#) feature is not enabled, `conditionalRenderingEnable` **must** be `VK_FALSE`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_CONDITIONAL_RENDERING_INFO_EXT`

Once recording starts, an application records a sequence of commands (`vkCmd*`) to set state in the command buffer, draw, dispatch, and other commands.

Several commands can also be recorded indirectly from `VkBuffer` content, see [Device-Generated Commands](#).

To complete recording of a command buffer, call:

```
VkResult vkEndCommandBuffer(  
    VkCommandBuffer  
        commandBuffer);
```

- `commandBuffer` is the command buffer to complete recording.

If there was an error during recording, the application will be notified by an unsuccessful return code returned by `vkEndCommandBuffer`. If the application wishes to further use the command buffer, the command buffer **must** be reset. The command buffer **must** have been in the `recording state`, and is moved to the `executable state`.

## Valid Usage

- `commandBuffer` **must** be in the `recording state`.
- If `commandBuffer` is a primary command buffer, there **must** not be an active render pass instance
- All queries made `active` during the recording of `commandBuffer` **must** have been made inactive
- Conditional rendering must not be `active`
- If `commandBuffer` is a secondary command buffer, there **must** not be an outstanding `vkCmdBeginDebugUtilsLabelEXT` command recorded to `commandBuffer` that has not previously been ended by a call to `vkCmdEndDebugUtilsLabelEXT`.
- If `commandBuffer` is a secondary command buffer, there **must** not be an outstanding `vkCmdDebugMarkerBeginEXT` command recorded to `commandBuffer` that has not previously been ended by a call to `vkCmdDebugMarkerEndEXT`.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

When a command buffer is in the executable state, it **can** be submitted to a queue for execution.

## 5.5. Command Buffer Submission

To submit command buffers to a queue, call:

```
VkResult vkQueueSubmit(  
    VkQueue queue,  
    uint32_t submitCount,  
    const VkSubmitInfo* pSubmits,  
    VkFence fence);
```

- `queue` is the queue that the command buffers will be submitted to.
- `submitCount` is the number of elements in the `pSubmits` array.
- `pSubmits` is a pointer to an array of `VkSubmitInfo` structures, each specifying a command buffer submission batch.
- `fence` is an **optional** handle to a fence to be signaled once all submitted command buffers have completed execution. If `fence` is not `VK_NULL_HANDLE`, it defines a [fence signal operation](#).

#### Note



Submission can be a high overhead operation, and applications **should** attempt to batch work together into as few calls to `vkQueueSubmit` as possible.

`vkQueueSubmit` is a [queue submission command](#), with each batch defined by an element of `pSubmits` as an instance of the `VkSubmitInfo` structure. Batches begin execution in the order they appear in `pSubmits`, but **may** complete out of order.

Fence and semaphore operations submitted with `vkQueueSubmit` have additional ordering constraints compared to other submission commands, with dependencies involving previous and subsequent queue operations. Information about these additional constraints can be found in the [semaphore](#) and [fence](#) sections of the [synchronization chapter](#).

Details on the interaction of `pWaitDstStageMask` with synchronization are described in the [semaphore wait operation](#) section of the [synchronization chapter](#).

The order that batches appear in `pSubmits` is used to determine [submission order](#), and thus all the

[implicit ordering guarantees](#) that respect it. Other than these implicit ordering guarantees and any [explicit synchronization primitives](#), these batches **may** overlap or otherwise execute out of order.

If any command buffer submitted to this queue is in the [executable state](#), it is moved to the [pending state](#). Once execution of all submissions of a command buffer complete, it moves from the [pending state](#), back to the [executable state](#). If a command buffer was recorded with the [VK\\_COMMAND\\_BUFFER\\_USAGE\\_ONE\\_TIME\\_SUBMIT\\_BIT](#) flag, it instead moves back to the [invalid state](#).

If `vkQueueSubmit` fails, it **may** return [VK\\_ERROR\\_OUT\\_OF\\_HOST\\_MEMORY](#) or [VK\\_ERROR\\_OUT\\_OF\\_DEVICE\\_MEMORY](#). If it does, the implementation **must** ensure that the state and contents of any resources or synchronization primitives referenced by the submitted command buffers and any semaphores referenced by `pSubmits` is unaffected by the call or its failure. If `vkQueueSubmit` fails in such a way that the implementation is unable to make that guarantee, the implementation **must** return [VK\\_ERROR\\_DEVICE\\_LOST](#). See [Lost Device](#).

## Valid Usage

- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be unsignaled
- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** not be associated with any other queue command that has not yet completed execution on that queue
- Any calls to `vkCmdSetEvent`, `vkCmdResetEvent` or `vkCmdWaitEvents` that have been recorded into any of the command buffer elements of the `pCommandBuffers` member of any element of `pSubmits`, **must** not reference any `VkEvent` that is referenced by any of those commands in a command buffer that has been submitted to another queue and is still in the *pending state*.
- Any stage flag included in any element of the `pWaitDstStageMask` member of any element of `pSubmits` **must** be a pipeline stage supported by one of the capabilities of `queue`, as specified in the [table of supported pipeline stages](#).
- Each element of the `pSignalSemaphores` member of any element of `pSubmits` **must** be unsignaled when the semaphore signal operation it defines is executed on the device
- When a semaphore wait operation referring to a binary semaphore defined by any element of the `pWaitSemaphores` member of any element of `pSubmits` executes on `queue`, there **must** be no other queues waiting on the same semaphore.
- All elements of the `pWaitSemaphores` member of all elements of `pSubmits` **must** be semaphores that are signaled, or have [semaphore signal operations](#) previously submitted for execution.
- All elements of the `pWaitSemaphores` member of all elements of `pSubmits` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR` **must** reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) **must** have also been submitted for execution.
- Each element of the `pCommandBuffers` member of each element of `pSubmits` **must** be in the [pending or executable state](#).
- If any element of the `pCommandBuffers` member of any element of `pSubmits` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`, it **must** not be in the *pending state*.
- Any [secondary command buffers recorded](#) into any element of the `pCommandBuffers` member of any element of `pSubmits` **must** be in the [pending or executable state](#).
- If any [secondary command buffers recorded](#) into any element of the `pCommandBuffers` member of any element of `pSubmits` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`, it **must** not be in the *pending state*.
- Each element of the `pCommandBuffers` member of each element of `pSubmits` **must** have been allocated from a `VkCommandPool` that was created for the same queue family `queue` belongs to.
- If any element of `pSubmits->pCommandBuffers` includes a [Queue Family Transfer Acquire Operation](#), there **must** exist a previously submitted [Queue Family Transfer Release Operation](#) on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such [acquire](#)

operations, and which happens before the acquire operation.

- If a command recorded into any element of `pCommandBuffers` was a `vkCmdBeginQuery` whose `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `profiling lock` **must** have been held continuously on the `VkDevice` that `queue` was retrieved from, throughout recording of those command buffers.
- Any resource created with `VK_SHARING_MODE_EXCLUSIVE` that is read by an operation specified by `pSubmits` **must** not be owned by any queue family other than the one which `queue` belongs to, at the time it is executed

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- If `submitCount` is not `0`, `pSubmits` **must** be a valid pointer to an array of `submitCount` valid `VkSubmitInfo` structures
- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be a valid `VkFence` handle
- Both of `fence`, and `queue` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `queue` **must** be externally synchronized
- Host access to `fence` **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	Any	-

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

The `VkSubmitInfo` structure is defined as:

```

typedef struct VkSubmitInfo {
    VkStructureType          sType;
    const void*             pNext;
    uint32_t                 waitSemaphoreCount;
    const VkSemaphore*      pWaitSemaphores;
    const VkPipelineStageFlags* pWaitDstStageMask;
    uint32_t                 commandBufferCount;
    const VkCommandBuffer*   pCommandBuffers;
    uint32_t                 signalSemaphoreCount;
    const VkSemaphore*      pSignalSemaphores;
} VkSubmitInfo;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **waitSemaphoreCount** is the number of semaphores upon which to wait before executing the command buffers for the batch.
- **pWaitSemaphores** is a pointer to an array of **VkSemaphore** handles upon which to wait before the command buffers for this batch begin execution. If semaphores to wait on are provided, they define a [semaphore wait operation](#).
- **pWaitDstStageMask** is a pointer to an array of pipeline stages at which each corresponding semaphore wait will occur.
- **commandBufferCount** is the number of command buffers to execute in the batch.
- **pCommandBuffers** is a pointer to an array of **VkCommandBuffer** handles to execute in the batch.
- **signalSemaphoreCount** is the number of semaphores to be signaled once the commands specified in **pCommandBuffers** have completed execution.
- **pSignalSemaphores** is a pointer to an array of **VkSemaphore** handles which will be signaled when the command buffers for this batch have completed execution. If semaphores to be signaled are provided, they define a [semaphore signal operation](#).

The order that command buffers appear in **pCommandBuffers** is used to determine [submission order](#), and thus all the [implicit ordering guarantees](#) that respect it. Other than these implicit ordering guarantees and any [explicit synchronization primitives](#), these command buffers **may** overlap or otherwise execute out of order.

## Valid Usage

- Each element of `pCommandBuffers` **must** not have been allocated with `VK_COMMAND_BUFFER_LEVEL_SECONDARY`
- If the `geometry shaders` feature is not enabled, each element of `pWaitDstStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, each element of `pWaitDstStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- Each element of `pWaitDstStageMask` **must** not include `VK_PIPELINE_STAGE_HOST_BIT`.
- If any element of `pWaitSemaphores` or `pSignalSemaphores` was created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` then the `pNext` chain **must** include an instance of `VkTimelineSemaphoreSubmitInfoKHR`
- If the `pNext` chain of this structure includes an instance of `VkTimelineSemaphoreSubmitInfoKHR` and any element of `pWaitSemaphores` was created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` then its `waitSemaphoreValueCount` member **must** equal `waitForSemaphoreCount`
- If the `pNext` chain of this structure includes an instance of `VkTimelineSemaphoreSubmitInfoKHR` and any element of `pSignalSemaphores` was created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` then its `signalSemaphoreValueCount` member **must** equal `signalSemaphoreCount`
- For each element of `pSignalSemaphores` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` the corresponding element of `VkTimelineSemaphoreSubmitInfoKHR::pSignalSemaphoreValues` **must** have a value greater than the current value of the semaphore when the `semaphore signal operation` is executed
- For each element of `pWaitSemaphores` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` the corresponding element of `VkTimelineSemaphoreSubmitInfoKHR::pWaitSemaphoreValues` **must** have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.
- For each element of `pSignalSemaphores` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` the corresponding element of `VkTimelineSemaphoreSubmitInfoKHR::pSignalSemaphoreValues` **must** have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.
- If the `mesh shaders` feature is not enabled, each element of `pWaitDstStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, each element of `pWaitDstStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SUBMIT_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkD3D12FenceSubmitInfoKHR`, `VkDeviceGroupSubmitInfo`, `VkProtectedSubmitInfo`, `VkWin32KeyedMutexAcquireReleaseInfoKHR`, `VkWin32KeyedMutexAcquireReleaseInfoNV` or `VkTimelineSemaphoreSubmitInfoKHR`,
- Each `sType` member in the `pNext` chain must be unique
- If `waitSemaphoreCount` is not `0`, `pWaitSemaphores` must be a valid pointer to an array of `waitSemaphoreCount` valid `VkSemaphore` handles
- If `waitSemaphoreCount` is not `0`, `pWaitDstStageMask` must be a valid pointer to an array of `waitSemaphoreCount` valid combinations of `VkPipelineStageFlagBits` values
- Each element of `pWaitDstStageMask` must not be `0`
- If `commandBufferCount` is not `0`, `pCommandBuffers` must be a valid pointer to an array of `commandBufferCount` valid `VkCommandBuffer` handles
- If `signalSemaphoreCount` is not `0`, `pSignalSemaphores` must be a valid pointer to an array of `signalSemaphoreCount` valid `VkSemaphore` handles
- Each of the elements of `pCommandBuffers`, the elements of `pSignalSemaphores`, and the elements of `pWaitSemaphores` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

To specify the values to use when waiting for and signaling semaphores created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR`, add the `VkTimelineSemaphoreSubmitInfoKHR` structure to the `pNext` chain of the `VkSubmitInfo` structure when using `vkQueueSubmit` or the `VkBindSparseInfo` structure when using `vkQueueBindSparse`. The `VkTimelineSemaphoreSubmitInfoKHR` structure is defined as:

```
typedef struct VkTimelineSemaphoreSubmitInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           waitSemaphoreValueCount;
    const uint64_t*    pWaitSemaphoreValues;
    uint32_t           signalSemaphoreValueCount;
    const uint64_t*    pSignalSemaphoreValues;
} VkTimelineSemaphoreSubmitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `waitSemaphoreValueCount` is the number of semaphore wait values specified in `pWaitSemaphoreValues`.

- `pWaitSemaphoreValues` is an array of length `waitSemaphoreValueCount` containing values for the corresponding semaphores in `VkSubmitInfo::pWaitSemaphores` to wait for.
- `signalSemaphoreValueCount` is the number of semaphore signal values specified in `pSignalSemaphoreValues`.
- `pSignalSemaphoreValues` is an array of length `signalSemaphoreValueCount` containing values for the corresponding semaphores in `VkSubmitInfo::pSignalSemaphores` to set when signaled.

If the semaphore in `VkSubmitInfo::pWaitSemaphores` or `VkSubmitInfo::pSignalSemaphores` corresponding to an entry in `pWaitSemaphoreValues` or `pSignalSemaphoreValues` respectively was not created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR`, the implementation **must** ignore the value in the `pWaitSemaphoreValues` or `pSignalSemaphoreValues` entry.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO_KHR`
- If `waitSemaphoreValueCount` is not `0`, and `pWaitSemaphoreValues` is not `NULL`, `pWaitSemaphoreValues` **must** be a valid pointer to an array of `waitSemaphoreValueCount` `uint64_t` values
- If `signalSemaphoreValueCount` is not `0`, and `pSignalSemaphoreValues` is not `NULL`, `pSignalSemaphoreValues` **must** be a valid pointer to an array of `signalSemaphoreValueCount` `uint64_t` values

To specify the values to use when waiting for and signaling semaphores whose [current payload](#) refers to a Direct3D 12 fence, add the `VkD3D12FenceSubmitInfoKHR` structure to the `pNext` chain of the `VkSubmitInfo` structure. The `VkD3D12FenceSubmitInfoKHR` structure is defined as:

```
typedef struct VkD3D12FenceSubmitInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreValuesCount;
    const uint64_t* pWaitSemaphoreValues;
    uint32_t signalSemaphoreValuesCount;
    const uint64_t* pSignalSemaphoreValues;
} VkD3D12FenceSubmitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `waitSemaphoreValuesCount` is the number of semaphore wait values specified in `pWaitSemaphoreValues`.
- `pWaitSemaphoreValues` is a pointer to an array of `waitSemaphoreValuesCount` values for the corresponding semaphores in `VkSubmitInfo::pWaitSemaphores` to wait for.
- `signalSemaphoreValuesCount` is the number of semaphore signal values specified in `pSignalSemaphoreValues`.

- `pSignalSemaphoreValues` is a pointer to an array of `signalSemaphoreValuesCount` values for the corresponding semaphores in `VkSubmitInfo::pSignalSemaphores` to set when signaled.

If the semaphore in `VkSubmitInfo::pWaitSemaphores` or `VkSubmitInfo::pSignalSemaphores` corresponding to an entry in `pWaitSemaphoreValues` or `pSignalSemaphoreValues` respectively does not currently have a `payload` referring to a Direct3D 12 fence, the implementation **must** ignore the value in the `pWaitSemaphoreValues` or `pSignalSemaphoreValues` entry.

*Note*

As the introduction of the external semaphore handle type `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT` predates that of timeline semaphores, support for importing semaphore payloads from external handles of that type into semaphores created (implicitly or explicitly) with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR` is preserved for backwards compatibility. However, applications **should** prefer importing such handle types into semaphores created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR`, and use the `VkTimelineSemaphoreSubmitInfoKHR` structure instead of the `VkD3D12FenceSubmitInfoKHR` structure to specify the values to use when waiting for and signaling such semaphores.



## Valid Usage

- `waitSemaphoreValuesCount` **must** be the same value as `VkSubmitInfo::waitSemaphoreCount`, where `VkSubmitInfo` is in the `pNext` chain of this `VkD3D12FenceSubmitInfoKHR` structure.
- `signalSemaphoreValuesCount` **must** be the same value as `VkSubmitInfo::signalSemaphoreCount`, where `VkSubmitInfo` is in the `pNext` chain of this `VkD3D12FenceSubmitInfoKHR` structure.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHR`
- If `waitSemaphoreValuesCount` is not `0`, and `pWaitSemaphoreValues` is not `NULL`, `pWaitSemaphoreValues` **must** be a valid pointer to an array of `waitSemaphoreValuesCount` `uint64_t` values
- If `signalSemaphoreValuesCount` is not `0`, and `pSignalSemaphoreValues` is not `NULL`, `pSignalSemaphoreValues` **must** be a valid pointer to an array of `signalSemaphoreValuesCount` `uint64_t` values

When submitting work that operates on memory imported from a Direct3D 11 resource to a queue, the keyed mutex mechanism **may** be used in addition to Vulkan semaphores to synchronize the work. Keyed mutexes are a property of a properly created shareable Direct3D 11 resource. They **can** only be used if the imported resource was created with the `D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX` flag.

To acquire keyed mutexes before submitted work and/or release them after, add a `VkWin32KeyedMutexAcquireReleaseInfoKHR` structure to the `pNext` chain of the `VkSubmitInfo` structure.

The `VkWin32KeyedMutexAcquireReleaseInfoKHR` structure is defined as:

```
typedef struct VkWin32KeyedMutexAcquireReleaseInfoKHR {
    VkStructureType         sType;
    const void*             pNext;
    uint32_t                acquireCount;
    const VkDeviceMemory*   pAcquireSyncs;
    const uint64_t*          pAcquireKeys;
    const uint32_t*          pAcquireTimeouts;
    uint32_t                releaseCount;
    const VkDeviceMemory*   pReleaseSyncs;
    const uint64_t*          pReleaseKeys;
} VkWin32KeyedMutexAcquireReleaseInfoKHR;
```

- `acquireCount` is the number of entries in the `pAcquireSyncs`, `pAcquireKeys`, and `pAcquireTimeoutMilliseconds` arrays.
- `pAcquireSyncs` is a pointer to an array of `VkDeviceMemory` objects which were imported from Direct3D 11 resources.
- `pAcquireKeys` is a pointer to an array of mutex key values to wait for prior to beginning the submitted work. Entries refer to the keyed mutex associated with the corresponding entries in `pAcquireSyncs`.
- `pAcquireTimeoutMilliseconds` is a pointer to an array of timeout values, in millisecond units, for each acquire specified in `pAcquireKeys`.
- `releaseCount` is the number of entries in the `pReleaseSyncs` and `pReleaseKeys` arrays.
- `pReleaseSyncs` is a pointer to an array of `VkDeviceMemory` objects which were imported from Direct3D 11 resources.
- `pReleaseKeys` is a pointer to an array of mutex key values to set when the submitted work has completed. Entries refer to the keyed mutex associated with the corresponding entries in `pReleaseSyncs`.

## Valid Usage

- Each member of `pAcquireSyncs` and `pReleaseSyncs` **must** be a device memory object imported by setting `VkImportMemoryWin32HandleInfoKHR::handleType` to `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHR`
- If `acquireCount` is not `0`, `pAcquireSyncs` **must** be a valid pointer to an array of `acquireCount` valid `VkDeviceMemory` handles
- If `acquireCount` is not `0`, `pAcquireKeys` **must** be a valid pointer to an array of `acquireCount` `uint64_t` values
- If `acquireCount` is not `0`, `pAcquireTimeouts` **must** be a valid pointer to an array of `acquireCount` `uint32_t` values
- If `releaseCount` is not `0`, `pReleaseSyncs` **must** be a valid pointer to an array of `releaseCount` valid `VkDeviceMemory` handles
- If `releaseCount` is not `0`, `pReleaseKeys` **must** be a valid pointer to an array of `releaseCount` `uint64_t` values
- Both of the elements of `pAcquireSyncs`, and the elements of `pReleaseSyncs` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

When submitting work that operates on memory imported from a Direct3D 11 resource to a queue, the keyed mutex mechanism **may** be used in addition to Vulkan semaphores to synchronize the work. Keyed mutexes are a property of a properly created shareable Direct3D 11 resource. They **can** only be used if the imported resource was created with the `D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX` flag.

To acquire keyed mutexes before submitted work and/or release them after, add a `VkWin32KeyedMutexAcquireReleaseInfoNV` structure to the `pNext` chain of the `VkSubmitInfo` structure.

The `VkWin32KeyedMutexAcquireReleaseInfoNV` structure is defined as:

```
typedef struct VkWin32KeyedMutexAcquireReleaseInfoNV {
    VkStructureType          sType;
    const void*              pNext;
    uint32_t                 acquireCount;
    const VkDeviceMemory*    pAcquireSyncs;
    const uint64_t*          pAcquireKeys;
    const uint32_t*          pAcquireTimeoutMilliseconds;
    uint32_t                 releaseCount;
    const VkDeviceMemory*    pReleaseSyncs;
    const uint64_t*          pReleaseKeys;
} VkWin32KeyedMutexAcquireReleaseInfoNV;
```

- `acquireCount` is the number of entries in the `pAcquireSyncs`, `pAcquireKeys`, and `pAcquireTimeoutMilliseconds` arrays.
- `pAcquireSyncs` is a pointer to an array of `VkDeviceMemory` objects which were imported from

Direct3D 11 resources.

- `pAcquireKeys` is a pointer to an array of mutex key values to wait for prior to beginning the submitted work. Entries refer to the keyed mutex associated with the corresponding entries in `pAcquireSyncs`.
- `pAcquireTimeoutMilliseconds` is a pointer to an array of timeout values, in millisecond units, for each acquire specified in `pAcquireKeys`.
- `releaseCount` is the number of entries in the `pReleaseSyncs` and `pReleaseKeys` arrays.
- `pReleaseSyncs` is a pointer to an array of `VkDeviceMemory` objects which were imported from Direct3D 11 resources.
- `pReleaseKeys` is a pointer to an array of mutex key values to set when the submitted work has completed. Entries refer to the keyed mutex associated with the corresponding entries in `pReleaseSyncs`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_NV`
- If `acquireCount` is not `0`, `pAcquireSyncs` **must** be a valid pointer to an array of `acquireCount` valid `VkDeviceMemory` handles
- If `acquireCount` is not `0`, `pAcquireKeys` **must** be a valid pointer to an array of `acquireCount` `uint64_t` values
- If `acquireCount` is not `0`, `pAcquireTimeoutMilliseconds` **must** be a valid pointer to an array of `acquireCount` `uint32_t` values
- If `releaseCount` is not `0`, `pReleaseSyncs` **must** be a valid pointer to an array of `releaseCount` valid `VkDeviceMemory` handles
- If `releaseCount` is not `0`, `pReleaseKeys` **must** be a valid pointer to an array of `releaseCount` `uint64_t` values
- Both of the elements of `pAcquireSyncs`, and the elements of `pReleaseSyncs` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

If the `pNext` chain of `VkSubmitInfo` includes a `VkProtectedSubmitInfo` structure, then the structure indicates whether the batch is protected. The `VkProtectedSubmitInfo` structure is defined as:

```
typedef struct VkProtectedSubmitInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkBool32           protectedSubmit;
} VkProtectedSubmitInfo;
```

- `protectedSubmit` specifies whether the batch is protected. If `protectedSubmit` is `VK_TRUE`, the batch is protected. If `protectedSubmit` is `VK_FALSE`, the batch is unprotected. If the `VkSubmitInfo::pNext` chain does not contain this structure, the batch is unprotected.

## Valid Usage

- If the protected memory feature is not enabled, `protectedSubmit` must not be `VK_TRUE`.
- If `protectedSubmit` is `VK_TRUE`, then each element of the `pCommandBuffers` array must be a protected command buffer.
- If `protectedSubmit` is `VK_FALSE`, then each element of the `pCommandBuffers` array must be an unprotected command buffer.
- If the `VkSubmitInfo::pNext` chain does not include a `VkProtectedSubmitInfo` structure, then each element of the command buffer of the `pCommandBuffers` array must be an unprotected command buffer.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO`

If the `pNext` chain of `VkSubmitInfo` includes a `VkDeviceGroupSubmitInfo` structure, then that structure includes device indices and masks specifying which physical devices execute semaphore operations and command buffers.

The `VkDeviceGroupSubmitInfo` structure is defined as:

```
typedef struct VkDeviceGroupSubmitInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           waitSemaphoreCount;
    const uint32_t*    pWaitSemaphoreDeviceIndices;
    uint32_t           commandBufferCount;
    const uint32_t*    pCommandBufferDeviceMasks;
    uint32_t           signalSemaphoreCount;
    const uint32_t*    pSignalSemaphoreDeviceIndices;
} VkDeviceGroupSubmitInfo;
```

or the equivalent

```
typedef VkDeviceGroupSubmitInfo VkDeviceGroupSubmitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `waitSemaphoreCount` is the number of elements in the `pWaitSemaphoreDeviceIndices` array.
- `pWaitSemaphoreDeviceIndices` is a pointer to an array of `waitSemaphoreCount` device indices indicating which physical device executes the semaphore wait operation in the corresponding element of `VkSubmitInfo::pWaitSemaphores`.

- `commandBufferCount` is the number of elements in the `pCommandBufferDeviceMasks` array.
- `pCommandBufferDeviceMasks` is a pointer to an array of `commandBufferCount` device masks indicating which physical devices execute the command buffer in the corresponding element of `VkSubmitInfo::pCommandBuffers`. A physical device executes the command buffer if the corresponding bit is set in the mask.
- `signalSemaphoreCount` is the number of elements in the `pSignalSemaphoreDeviceIndices` array.
- `pSignalSemaphoreDeviceIndices` is a pointer to an array of `signalSemaphoreCount` device indices indicating which physical device executes the semaphore signal operation in the corresponding element of `VkSubmitInfo::pSignalSemaphores`.

If this structure is not present, semaphore operations and command buffers execute on device index zero.

## Valid Usage

- `waitSemaphoreCount` **must** equal `VkSubmitInfo::waitSemaphoreCount`
- `commandBufferCount` **must** equal `VkSubmitInfo::commandBufferCount`
- `signalSemaphoreCount` **must** equal `VkSubmitInfo::signalSemaphoreCount`
- All elements of `pWaitSemaphoreDeviceIndices` and `pSignalSemaphoreDeviceIndices` **must** be valid device indices
- All elements of `pCommandBufferDeviceMasks` **must** be valid device masks

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO`
- If `waitSemaphoreCount` is not `0`, `pWaitSemaphoreDeviceIndices` **must** be a valid pointer to an array of `waitSemaphoreCount uint32_t` values
- If `commandBufferCount` is not `0`, `pCommandBufferDeviceMasks` **must** be a valid pointer to an array of `commandBufferCount uint32_t` values
- If `signalSemaphoreCount` is not `0`, `pSignalSemaphoreDeviceIndices` **must** be a valid pointer to an array of `signalSemaphoreCount uint32_t` values

If the `pNext` chain of `VkSubmitInfo` includes a `VkPerformanceQuerySubmitInfoKHR` structure, then the structure indicates which counter pass is active for the batch in that submit.

The `VkPerformanceQuerySubmitInfoKHR` structure is defined as:

```
typedef struct VkPerformanceQuerySubmitInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t counterPassIndex;
} VkPerformanceQuerySubmitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `counterPassIndex` specifies which counter pass index is active.

If the `VkSubmitInfo::pNext` chain does not contain this structure, the batch defaults to use counter pass index 0.

### Valid Usage

- `counterPassIndex` **must** be less than the number of counter passes required by any queries within the batch. The required number of counter passes for a performance query is obtained by calling `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR`

## 5.6. Queue Forward Progress

When using binary semaphores, the application **must** ensure that command buffer submissions will be able to complete without any subsequent operations by the application on any queue. After any call to `vkQueueSubmit` (or other queue operation), for every queued wait on a semaphore created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR` there **must** be a prior signal of that semaphore that will not be consumed by a different wait on the semaphore.

When using timeline semaphores, wait-before-signal behavior is well-defined and applications **can** submit work via `vkQueueSubmit` which defines a `timeline semaphore wait operation` before submitting a corresponding `semaphore signal operation`. For each `timeline semaphore wait operation` defined by a call to `vkQueueSubmit`, the application **must** ensure that a corresponding `semaphore signal operation` is executed before forward progress can be made.

Command buffers in the submission **can** include `vkCmdWaitEvents` commands that wait on events that will not be signaled by earlier commands in the queue. Such events **must** be signaled by the application using `vkSetEvent`, and the `vkCmdWaitEvents` commands that wait upon them **must** not be inside a render pass instance. The event **must** be set before the `vkCmdWaitEvents` command is executed.

*Note*



Implementations may have some tolerance for waiting on events to be set, but this is defined outside of the scope of Vulkan.

## 5.7. Secondary Command Buffer Execution

A secondary command buffer **must** not be directly submitted to a queue. Instead, secondary command buffers are recorded to execute as part of a primary command buffer with the command:

```
void vkCmdExecuteCommands(  
    VkCommandBuffer  
    uint32_t  
    const VkCommandBuffer*  
                                commandBuffer,  
                                commandBufferCount,  
                                pCommandBuffers);
```

- `commandBuffer` is a handle to a primary command buffer that the secondary command buffers are executed in.
- `commandBufferCount` is the length of the `pCommandBuffers` array.
- `pCommandBuffers` is a pointer to an array of `commandBufferCount` secondary command buffer handles, which are recorded to execute in the primary command buffer in the order they are listed in the array.

If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, and it was recorded into any other primary command buffer which is currently in the [executable or recording state](#), that primary command buffer becomes [invalid](#).

## Valid Usage

- `commandBuffer` **must** have been allocated with a `level` of `VK_COMMAND_BUFFER_LEVEL_PRIMARY`
- Each element of `pCommandBuffers` **must** have been allocated with a `level` of `VK_COMMAND_BUFFER_LEVEL_SECONDARY`
- Each element of `pCommandBuffers` **must** be in the `pending` or `executable` state.
- If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, and it was recorded into any other primary command buffer, that primary command buffer **must** not be in the `pending` state
- If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it **must** not be in the `pending` state.
- If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it **must** not have already been recorded to `commandBuffer`.
- If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it **must** not appear more than once in `pCommandBuffers`.
- Each element of `pCommandBuffers` **must** have been allocated from a `VkCommandPool` that was created for the same queue family as the `VkCommandPool` from which `commandBuffer` was allocated
- If `vkCmdExecuteCommands` is being called within a render pass instance, that render pass instance **must** have been begun with the `contents` parameter of `vkCmdBeginRenderPass` set to `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS`
- If `vkCmdExecuteCommands` is being called within a render pass instance, each element of `pCommandBuffers` **must** have been recorded with the `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`
- If `vkCmdExecuteCommands` is being called within a render pass instance, each element of `pCommandBuffers` **must** have been recorded with `VkCommandBufferInheritanceInfo::subpass` set to the index of the subpass which the given command buffer will be executed in
- If `vkCmdExecuteCommands` is being called within a render pass instance, the render passes specified in the `pBeginInfo::pInheritanceInfo::renderPass` members of the `vkBeginCommandBuffer` commands used to begin recording each element of `pCommandBuffers` **must** be `compatible` with the current render pass.
- If `vkCmdExecuteCommands` is being called within a render pass instance, and any element of `pCommandBuffers` was recorded with `VkCommandBufferInheritanceInfo::framebuffer` not equal to `VK_NULL_HANDLE`, that `VkFramebuffer` **must** match the `VkFramebuffer` used in the current render pass instance
- If `vkCmdExecuteCommands` is not being called within a render pass instance, each element of `pCommandBuffers` **must** not have been recorded with the `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`
- If the `inherited queries` feature is not enabled, `commandBuffer` **must** not have any queries active

- If `commandBuffer` has a `VK_QUERY_TYPE_OCCLUSION` query `active`, then each element of `pCommandBuffers` **must** have been recorded with `VkCommandBufferInheritanceInfo` `::occlusionQueryEnable` set to `VK_TRUE`
- If `commandBuffer` has a `VK_QUERY_TYPE_OCCLUSION` query `active`, then each element of `pCommandBuffers` **must** have been recorded with `VkCommandBufferInheritanceInfo` `::queryFlags` having all bits set that are set for the query
- If `commandBuffer` has a `VK_QUERY_TYPE_PIPELINE_STATISTICS` query `active`, then each element of `pCommandBuffers` **must** have been recorded with `VkCommandBufferInheritanceInfo` `::pipelineStatistics` having all bits set that are set in the `VkQueryPool` the query uses
- Each element of `pCommandBuffers` **must** not begin any query types that are `active` in `commandBuffer`
- If `commandBuffer` is a protected command buffer, then each element of `pCommandBuffers` **must** be a protected command buffer.
- If `commandBuffer` is an unprotected command buffer, then each element of `pCommandBuffers` **must** be an unprotected command buffer.
- This command **must** not be recorded when transform feedback is active

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pCommandBuffers` **must** be a valid pointer to an array of `commandBufferCount` valid `VkCommandBuffer` handles
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- `commandBuffer` **must** be a primary `VkCommandBuffer`
- `commandBufferCount` **must** be greater than `0`
- Both of `commandBuffer`, and the elements of `pCommandBuffers` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Both	Transfer Graphics Compute	

## 5.8. Command Buffer Device Mask

Each command buffer has a piece of state storing the current device mask of the command buffer. This mask controls which physical devices within the logical device all subsequent commands will execute on, including state-setting commands, action commands, and synchronization commands.

Scissor, exclusive scissor, and viewport state **can** be set to different values on each physical device (only when set as dynamic state), and each physical device will render using its local copy of the state. Other state is shared between physical devices, such that all physical devices use the most recently set values for the state. However, when recording an action command that uses a piece of state, the most recent command that set that state **must** have included all physical devices that execute the action command in its current device mask.

The command buffer's device mask is orthogonal to the `pCommandBufferDeviceMasks` member of `VkDeviceGroupSubmitInfo`. Commands only execute on a physical device if the device index is set in both device masks.

If the `pNext` chain of `VkCommandBufferBeginInfo` includes a `VkDeviceGroupCommandBufferBeginInfo` structure, then that structure includes an initial device mask for the command buffer.

The `VkDeviceGroupCommandBufferBeginInfo` structure is defined as:

```
typedef struct VkDeviceGroupCommandBufferBeginInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           deviceMask;
} VkDeviceGroupCommandBufferBeginInfo;
```

or the equivalent

```
typedef VkDeviceGroupCommandBufferBeginInfo VkDeviceGroupCommandBufferBeginInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `deviceMask` is the initial value of the command buffer's device mask.

The initial device mask also acts as an upper bound on the set of devices that **can** ever be in the

device mask in the command buffer.

If this structure is not present, the initial value of a command buffer's device mask is set to include all physical devices in the logical device when the command buffer begins recording.

## Valid Usage

- `deviceMask` **must** be a valid device mask value
- `deviceMask` **must** not be zero

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO`

To update the current device mask of a command buffer, call:

```
void vkCmdSetDeviceMask(  
    VkCommandBuffer  
    uint32_t  
        commandBuffer,  
        deviceMask);
```

or the equivalent command

```
void vkCmdSetDeviceMaskKHR(  
    VkCommandBuffer  
    uint32_t  
        commandBuffer,  
        deviceMask);
```

- `commandBuffer` is command buffer whose current device mask is modified.
- `deviceMask` is the new value of the current device mask.

`deviceMask` is used to filter out subsequent commands from executing on all physical devices whose bit indices are not set in the mask, except commands beginning a render pass instance, commands transitioning to the next subpass in the render pass instance, and commands ending a render pass instance, which always execute on the set of physical devices whose bit indices are included in the `deviceMask` member of the instance of the `VkDeviceGroupRenderPassBeginInfo` structure passed to the command beginning the corresponding render pass instance.

## Valid Usage

- `deviceMask` **must** be a valid device mask value
- `deviceMask` **must** not be zero
- `deviceMask` **must** not include any set bits that were not in the `VkDeviceGroupCommandBufferBeginInfo::deviceMask` value when the command buffer began recording.
- If `vkCmdSetDeviceMask` is called inside a render pass instance, `deviceMask` **must** not include any set bits that were not in the `VkDeviceGroupRenderPassBeginInfo::deviceMask` value when the render pass instance began recording.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, compute, or transfer operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute Transfer	

# Chapter 6. Synchronization and Cache Control

Synchronization of access to resources is primarily the responsibility of the application in Vulkan. The order of execution of commands with respect to the host and other commands on the device has few implicit guarantees, and needs to be explicitly specified. Memory caches and other optimizations are also explicitly managed, requiring that the flow of data through the system is largely under application control.

Whilst some implicit guarantees exist between commands, five explicit synchronization mechanisms are exposed by Vulkan:

## Fences

Fences **can** be used to communicate to the host that execution of some task on the device has completed.

## Semaphores

Semaphores **can** be used to control resource access across multiple queues.

## Events

Events provide a fine-grained synchronization primitive which **can** be signaled either within a command buffer or by the host, and **can** be waited upon within a command buffer or queried on the host.

## Pipeline Barriers

Pipeline barriers also provide synchronization control within a command buffer, but at a single point, rather than with separate signal and wait operations.

## Render Passes

Render passes provide a useful synchronization framework for most rendering tasks, built upon the concepts in this chapter. Many cases that would otherwise need an application to use other synchronization primitives **can** be expressed more efficiently as part of a render pass.

## 6.1. Execution and Memory Dependencies

An *operation* is an arbitrary amount of work to be executed on the host, a device, or an external entity such as a presentation engine. Synchronization commands introduce explicit *execution dependencies*, and *memory dependencies* between two sets of operations defined by the command's two *synchronization scopes*.

The synchronization scopes define which other operations a synchronization command is able to create execution dependencies with. Any type of operation that is not in a synchronization command's synchronization scopes will not be included in the resulting dependency. For example, for many synchronization commands, the synchronization scopes **can** be limited to just operations executing in specific [pipeline stages](#), which allows other pipeline stages to be excluded from a dependency. Other scoping options are possible, depending on the particular command.

An *execution dependency* is a guarantee that for two sets of operations, the first set **must happen-before** the second set. If an operation happens-before another operation, then the first operation **must complete** before the second operation is initiated. More precisely:

- Let **A** and **B** be separate sets of operations.
- Let **S** be a synchronization command.
- Let **A<sub>s</sub>** and **B<sub>s</sub>** be the synchronization scopes of **S**.
- Let **A'** be the intersection of sets **A** and **A<sub>s</sub>**.
- Let **B'** be the intersection of sets **B** and **B<sub>s</sub>**.
- Submitting **A**, **S** and **B** for execution, in that order, will result in execution dependency **E** between **A'** and **B'**.
- Execution dependency **E** guarantees that **A'** happens-before **B'**.

An *execution dependency chain* is a sequence of execution dependencies that form a happens-before relation between the first dependency's **A'** and the final dependency's **B'**. For each consecutive pair of execution dependencies, a chain exists if the intersection of **B<sub>s</sub>** in the first dependency and **A<sub>s</sub>** in the second dependency is not an empty set. The formation of a single execution dependency from an execution dependency chain can be described by substituting the following in the description of execution dependencies:

- Let **S** be a set of synchronization commands that generate an execution dependency chain.
- Let **A<sub>s</sub>** be the first synchronization scope of the first command in **S**.
- Let **B<sub>s</sub>** be the second synchronization scope of the last command in **S**.

Execution dependencies alone are not sufficient to guarantee that values resulting from writes in one set of operations **can** be read from another set of operations.

Three additional types of operation are used to control memory access. *Availability operations* cause the values generated by specified memory write accesses to become *available* to a memory domain for future access. Any available value remains available until a subsequent write to the same memory location occurs (whether it is made available or not) or the memory is freed. *Memory domain operations* cause writes that are available to a source memory domain to become available to a destination memory domain (an example of this is making writes available to the host domain available to the device domain). *Visibility operations* cause values available to a memory domain to become *visible* to specified memory accesses.

Availability, visibility, memory domains, and memory domain operations are formally defined in the [Availability and Visibility](#) section of the [Memory Model](#) chapter. Which API operations perform each of these operations is defined in [Availability, Visibility, and Domain Operations](#).

A *memory dependency* is an execution dependency which includes availability and visibility operations such that:

- The first set of operations happens-before the availability operation.
- The availability operation happens-before the visibility operation.
- The visibility operation happens-before the second set of operations.

Once written values are made visible to a particular type of memory access, they **can** be read or written by that type of memory access. Most synchronization commands in Vulkan define a memory dependency.

The specific memory accesses that are made available and visible are defined by the *access scopes* of a memory dependency. Any type of access that is in a memory dependency's first access scope and occurs in **A'** is made available. Any type of access that is in a memory dependency's second access scope and occurs in **B'** has any available writes made visible to it. Any type of operation that is not in a synchronization command's access scopes will not be included in the resulting dependency.

A memory dependency enforces availability and visibility of memory accesses and execution order between two sets of operations. Adding to the description of [execution dependency chains](#):

- Let **a** be the set of memory accesses performed by **A'**.
- Let **b** be the set of memory accesses performed by **B'**.
- Let **a<sub>s</sub>** be the first access scope of the first command in **S**.
- Let **b<sub>s</sub>** be the second access scope of the last command in **S**.
- Let **a'** be the intersection of sets **a** and **a<sub>s</sub>**.
- Let **b'** be the intersection of sets **b** and **b<sub>s</sub>**.
- Submitting **A**, **S** and **B** for execution, in that order, will result in a memory dependency **m** between **A'** and **B'**.
- Memory dependency **m** guarantees that:
  - Memory writes in **a'** are made available.
  - Available memory writes, including those from **a'**, are made visible to **b'**.

*Note*

Execution and memory dependencies are used to solve data hazards, i.e. to ensure that read and write operations occur in a well-defined order. Write-after-read hazards can be solved with just an execution dependency, but read-after-write and write-after-write hazards need appropriate memory dependencies to be included between them. If an application does not include dependencies to solve these hazards, the results and execution orders of memory accesses are undefined.



### 6.1.1. Image Layout Transitions

Image subresources **can** be transitioned from one [layout](#) to another as part of a [memory dependency](#) (e.g. by using an [image memory barrier](#)). When a layout transition is specified in a memory dependency, it happens-after the availability operations in the memory dependency, and happens-before the visibility operations. Image layout transitions **may** perform read and write accesses on all memory bound to the image subresource range, so applications **must** ensure that all memory writes have been made [available](#) before a layout transition is executed. Available memory is automatically made visible to a layout transition, and writes performed by a layout transition are automatically made available.

Layout transitions always apply to a particular image subresource range, and specify both an old layout and new layout. If the old layout does not match the new layout, a transition occurs. The old layout **must** match the current layout of the image subresource range, with one exception. The old layout **can** always be specified as `VK_IMAGE_LAYOUT_UNDEFINED`, though doing so invalidates the contents of the image subresource range.

As image layout transitions **may** perform read and write accesses on the memory bound to the image, if the image subresource affected by the layout transition is bound to peer memory for any device in the current device mask then the memory heap the bound memory comes from **must** support the `VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT` and `VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT` capabilities as returned by `vkGetDeviceGroupPeerMemoryFeatures`.

*Note*



Setting the old layout to `VK_IMAGE_LAYOUT_UNDEFINED` implies that the contents of the image subresource need not be preserved. Implementations **may** use this information to avoid performing expensive data transition operations.

*Note*



Applications **must** ensure that layout transitions happen-after all operations accessing the image with the old layout, and happen-before any operations that will access the image with the new layout. Layout transitions are potentially read/write operations, so not defining appropriate memory dependencies to guarantee this will result in a data race.

Image layout transitions interact with [memory aliasing](#).

### 6.1.2. Pipeline Stages

The work performed by an [action or synchronization command](#) consists of multiple operations, which are performed as a sequence of logically independent steps known as *pipeline stages*. The exact pipeline stages executed depend on the particular command that is used, and current command buffer state when the command was recorded. [Drawing commands](#), [dispatching commands](#), [copy commands](#), [clear commands](#), and [synchronization commands](#) all execute in different sets of [pipeline stages](#). [Synchronization commands](#) do not execute in a defined pipeline, but do execute `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` and `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`.

*Note*



Operations performed by synchronization commands (e.g. [availability and visibility operations](#)) are not executed by a defined pipeline stage. However other commands can still synchronize with them via the `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` and `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` pipeline stages.

Execution of operations across pipeline stages **must** adhere to [implicit ordering guarantees](#), particularly including [pipeline stage order](#). Otherwise, execution across pipeline stages **may** overlap or execute out of order with regards to other stages, unless otherwise enforced by an execution dependency.

Several of the synchronization commands include pipeline stage parameters, restricting the [synchronization scopes](#) for that command to just those stages. This allows fine grained control over the exact execution dependencies and accesses performed by action commands. Implementations **should** use these pipeline stages to avoid unnecessary stalls or cache flushing.

Bits which can be set, specifying pipeline stages, are:

```
typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
    VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT = 0x01000000,
    VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000,
    VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX = 0x00020000,
    VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00400000,
    VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV = 0x00200000,
    VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV = 0x02000000,
    VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV = 0x00080000,
    VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV = 0x00100000,
    VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT = 0x00800000,
    VK_PIPELINE_STAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPipelineStageFlagBits;
```

- [VK\\_PIPELINE\\_STAGE\\_TOP\\_OF\\_PIPE\\_BIT](#) specifies the stage of the pipeline where any commands are initially received by the queue.
- [VK\\_PIPELINE\\_STAGE\\_DRAW\\_INDIRECT\\_BIT](#) specifies the stage of the pipeline where Draw/DispatchIndirect data structures are consumed. This stage also includes reading commands written by [vkCmdProcessCommandsNVX](#).
- [VK\\_PIPELINE\\_STAGE\\_TASK\\_SHADER\\_BIT\\_NV](#) specifies the task shader stage.
- [VK\\_PIPELINE\\_STAGE\\_MESH\\_SHADER\\_BIT\\_NV](#) specifies the mesh shader stage.
- [VK\\_PIPELINE\\_STAGE\\_VERTEX\\_INPUT\\_BIT](#) specifies the stage of the pipeline where vertex and index buffers are consumed.
- [VK\\_PIPELINE\\_STAGE\\_VERTEX\\_SHADER\\_BIT](#) specifies the vertex shader stage.

- `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` specifies the tessellation control shader stage.
- `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT` specifies the tessellation evaluation shader stage.
- `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT` specifies the geometry shader stage.
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT` specifies the fragment shader stage.
- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT` specifies the stage of the pipeline where early fragment tests (depth and stencil tests before fragment shading) are performed. This stage also includes [subpass load operations](#) for framebuffer attachments with a depth/stencil format.
- `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT` specifies the stage of the pipeline where late fragment tests (depth and stencil tests after fragment shading) are performed. This stage also includes [subpass store operations](#) for framebuffer attachments with a depth/stencil format.
- `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` specifies the stage of the pipeline after blending where the final color values are output from the pipeline. This stage also includes [subpass load and store operations](#) and multisample resolve operations for framebuffer attachments with a color or depth/stencil format.
- `VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT` specifies the execution of a compute shader.
- `VK_PIPELINE_STAGE_TRANSFER_BIT` specifies the execution of copy commands. This includes the operations resulting from all [copy commands](#), [clear commands](#) (with the exception of `vkCmdClearAttachments`), and `vkCmdCopyQueryPoolResults`.
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` specifies the final stage in the pipeline where operations generated by all commands complete execution.
- `VK_PIPELINE_STAGE_HOST_BIT` specifies a pseudo-stage indicating execution on the host of reads/writes of device memory. This stage is not invoked by any commands recorded in a command buffer.
- `VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV` specifies the execution of the ray tracing shader stages.
- `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV` specifies the execution of `vkCmdBuildAccelerationStructureNV`, `vkCmdCopyAccelerationStructureNV`, and `vkCmdWriteAccelerationStructuresPropertiesNV`.
- `VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT` specifies the execution of all graphics pipeline stages, and is equivalent to the logical OR of:
  - `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`
  - `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`
  - `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
  - `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
  - `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT`
  - `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`
  - `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`
  - `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
  - `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`
  - `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`
  - `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`
  - `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`
  - `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`
  - `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`
  - `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`
  - `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`
  - `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`
- `VK_PIPELINE_STAGE_ALL_COMMANDS_BIT` is equivalent to the logical OR of every other pipeline stage flag that is supported on the queue it is used with.
  - `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT` specifies the stage of the pipeline where the predicate of conditional rendering is consumed.
  - `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT` specifies the stage of the pipeline where vertex attribute output values are written to the transform feedback buffers.
  - `VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX` specifies the stage of the pipeline where device-side generation of commands via `vkCmdProcessCommandsNVX` is handled.
  - `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV` specifies the stage of the pipeline where the *shading rate image* is read to determine the shading rate for portions of a rasterized primitive.
  - `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT` specifies the stage of the pipeline where the fragment density map is read to *generate the fragment areas*.

*Note*

An execution dependency with only `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` in the destination stage mask will only prevent that stage from executing in subsequently submitted commands. As this stage does not perform any actual execution, this is not observable - in effect, it does not delay processing of subsequent commands. Similarly an execution dependency with only `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` in the source stage mask will effectively not wait for any prior commands to complete.



When defining a memory dependency, using only `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` or `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` would never make any accesses available and/or visible because these stages do not access memory.

`VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` and `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` are useful for accomplishing layout transitions and queue ownership operations when the required execution dependency is satisfied by other means - for example, semaphore operations between queues.

```
typedef VkFlags VkPipelineStageFlags;
```

`VkPipelineStageFlags` is a bitmask type for setting a mask of zero or more `VkPipelineStageFlagBits`.

If a synchronization command includes a source stage mask, its first [synchronization scope](#) only includes execution of the pipeline stages specified in that mask, and its first [access scope](#) only includes memory access performed by pipeline stages specified in that mask. If a synchronization command includes a destination stage mask, its second [synchronization scope](#) only includes execution of the pipeline stages specified in that mask, and its second [access scope](#) only includes memory access performed by pipeline stages specified in that mask.

*Note*

Including a particular pipeline stage in the first [synchronization scope](#) of a command implicitly includes [logically earlier](#) pipeline stages in the synchronization scope. Similarly, the second [synchronization scope](#) includes [logically later](#) pipeline stages.



However, note that [access scopes](#) are not affected in this way - only the precise stages specified are considered part of each access scope.

Certain pipeline stages are only available on queues that support a particular set of operations. The following table lists, for each pipeline stage flag, which queue capability flag **must** be supported by the queue. When multiple flags are enumerated in the second column of the table, it means that the pipeline stage is supported on the queue if it supports any of the listed capability flags. For further details on queue capabilities see [Physical Device Enumeration](#) and [Queues](#).

*Table 3. Supported pipeline stage flags*

Pipeline stage flag	Required queue capability flag
<code>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</code>	None required
<code>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code> or <code>VK_QUEUE_COMPUTE_BIT</code>
<code>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</code>	<code>VK_QUEUE_COMPUTE_BIT</code>
<code>VK_PIPELINE_STAGE_TRANSFER_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code> , <code>VK_QUEUE_COMPUTE_BIT</code> or <code>VK_QUEUE_TRANSFER_BIT</code>
<code>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</code>	None required
<code>VK_PIPELINE_STAGE_HOST_BIT</code>	None required
<code>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</code>	None required

Pipeline stage flag	Required queue capability flag
<code>VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code> or <code>VK_QUEUE_COMPUTE_BIT</code>
<code>VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX</code>	<code>VK_QUEUE_GRAPHICS_BIT</code> or <code>VK_QUEUE_COMPUTE_BIT</code>
<code>VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>
<code>VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV</code>	<code>VK_QUEUE_COMPUTE_BIT</code>
<code>VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV</code>	<code>VK_QUEUE_COMPUTE_BIT</code>
<code>VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT</code>	<code>VK_QUEUE_GRAPHICS_BIT</code>

Pipeline stages that execute as a result of a command logically complete execution in a specific order, such that completion of a logically later pipeline stage **must** not happen-before completion of a logically earlier stage. This means that including any stage in the source stage mask for a particular synchronization command also implies that any logically earlier stages are included in **A<sub>s</sub>** for that command.

Similarly, initiation of a logically earlier pipeline stage **must** not happen-after initiation of a logically later pipeline stage. Including any given stage in the destination stage mask for a particular synchronization command also implies that any logically later stages are included in **B<sub>s</sub>** for that command.

*Note*

Implementations **may** not support synchronization at every pipeline stage for every synchronization operation. If a pipeline stage that an implementation does not support synchronization for appears in a source stage mask, it **may** substitute any logically later stage in its place for the first synchronization scope. If a pipeline stage that an implementation does not support synchronization for appears in a destination stage mask, it **may** substitute any logically earlier stage in its place for the second synchronization scope.



For example, if an implementation is unable to signal an event immediately after vertex shader execution is complete, it **may** instead signal the event after color attachment output has completed.

If an implementation makes such a substitution, it **must** not affect the semantics of execution or memory dependencies or image and buffer memory barriers.

The order and set of pipeline stages executed by a given command is determined by the command's pipeline type, as described below:

For the graphics primitive shading pipeline, the following stages occur in this order:

- `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`
- `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`
- `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT`

- `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`
- `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`
- `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`
- `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`
- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`
- `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`

For the graphics mesh shading pipeline, the following stages occur in this order:

- `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`
- `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`
- `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
- `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`
- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`
- `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`

For graphics pipeline commands executing in a render pass with a fragment density map attachment, the pipeline stage where the fragment density map read happens has no particular order relative to the other stages except that it happens before `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`.

- `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

For the compute pipeline, the following stages occur in this order:

- `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`
- `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`
- `VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT`
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`

The conditional rendering stage is formally part of both the graphics, and the compute pipeline. The pipeline stage where the predicate read happens has unspecified order relative to other stages of these pipelines:

- `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`

For the transfer pipeline, the following stages occur in this order:

- `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`

- `VK_PIPELINE_STAGE_TRANSFER_BIT`
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`

For host operations, only one pipeline stage occurs, so no order is guaranteed:

- `VK_PIPELINE_STAGE_HOST_BIT`

For the command processing pipeline, the following stages occur in this order:

- `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`
- `VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX`
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`

For the ray tracing shader pipeline, only one pipeline stage occurs, so no order is guaranteed:

- `VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV`

For ray tracing acceleration structure operations, only one pipeline stage occurs, so no order is guaranteed:

- `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV`

### 6.1.3. Access Types

Memory in Vulkan **can** be accessed from within shader invocations and via some fixed-function stages of the pipeline. The *access type* is a function of the [descriptor type](#) used, or how a fixed-function stage accesses memory. Each access type corresponds to a bit flag in [VkAccessFlagBits](#).

Some synchronization commands take sets of access types as parameters to define the [access scopes](#) of a memory dependency. If a synchronization command includes a source access mask, its first [access scope](#) only includes accesses via the access types specified in that mask. Similarly, if a synchronization command includes a destination access mask, its second [access scope](#) only includes accesses via the access types specified in that mask.

Access types that **can** be set in an access mask include:

```

typedef enum VkAccessFlagBits {
    VK_ACCESS_INDIRECT_COMMAND_READ_BIT = 0x00000001,
    VK_ACCESS_INDEX_READ_BIT = 0x00000002,
    VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT = 0x00000004,
    VK_ACCESS_UNIFORM_READ_BIT = 0x00000008,
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT = 0x00000010,
    VK_ACCESS_SHADER_READ_BIT = 0x00000020,
    VK_ACCESS_SHADER_WRITE_BIT = 0x00000040,
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT = 0x00000080,
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT = 0x00000100,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT = 0x00000200,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT = 0x00000400,
    VK_ACCESS_TRANSFER_READ_BIT = 0x00000800,
    VK_ACCESS_TRANSFER_WRITE_BIT = 0x00001000,
    VK_ACCESS_HOST_READ_BIT = 0x00002000,
    VK_ACCESS_HOST_WRITE_BIT = 0x00004000,
    VK_ACCESS_MEMORY_READ_BIT = 0x00008000,
    VK_ACCESS_MEMORY_WRITE_BIT = 0x00010000,
    VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT = 0x02000000,
    VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT = 0x04000000,
    VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT = 0x08000000,
    VK_ACCESS_CONDITIONAL_RENDERING_READ_BIT_EXT = 0x00100000,
    VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX = 0x00020000,
    VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX = 0x00040000,
    VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT = 0x00080000,
    VK_ACCESS_SHADING_RATE_IMAGE_READ_BIT_NV = 0x00800000,
    VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NV = 0x00200000,
    VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NV = 0x00400000,
    VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT = 0x01000000,
    VK_ACCESS_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkAccessFlagBits;

```

- **VK\_ACCESS\_INDIRECT\_COMMAND\_READ\_BIT** specifies read access to indirect command data read as part of an indirect drawing or dispatch command.
- **VK\_ACCESS\_INDEX\_READ\_BIT** specifies read access to an index buffer as part of an indexed drawing command, bound by [vkCmdBindIndexBuffer](#).
- **VK\_ACCESS\_VERTEX\_ATTRIBUTE\_READ\_BIT** specifies read access to a vertex buffer as part of a drawing command, bound by [vkCmdBindVertexBuffers](#).
- **VK\_ACCESS\_UNIFORM\_READ\_BIT** specifies read access to a [uniform buffer](#).
- **VK\_ACCESS\_INPUT\_ATTACHMENT\_READ\_BIT** specifies read access to an [input attachment](#) within a render pass during fragment shading.
- **VK\_ACCESS\_SHADER\_READ\_BIT** specifies read access to a [storage buffer](#), [physical storage buffer](#), [uniform texel buffer](#), [storage texel buffer](#), [sampled image](#), or [storage image](#).
- **VK\_ACCESS\_SHADER\_WRITE\_BIT** specifies write access to a [storage buffer](#), [physical storage buffer](#), [storage texel buffer](#), or [storage image](#).
- **VK\_ACCESS\_COLOR\_ATTACHMENT\_READ\_BIT** specifies read access to a [color attachment](#), such as via

blending, logic operations, or via certain [subpass load operations](#). It does not include advanced blend operations.

- `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT` specifies write access to a [color](#), [resolve](#), or [depth/stencil resolve attachment](#) during a [render pass](#) or via certain [subpass load and store operations](#).
- `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT` specifies read access to a [depth/stencil attachment](#), via [depth](#) or [stencil operations](#) or via certain [subpass load and store operations](#).
- `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT` specifies write access to a [depth/stencil attachment](#), via [depth](#) or [stencil operations](#) or via certain [subpass load and store operations](#).
- `VK_ACCESS_TRANSFER_READ_BIT` specifies read access to an image or buffer in a [copy](#) operation.
- `VK_ACCESS_TRANSFER_WRITE_BIT` specifies write access to an image or buffer in a [clear](#) or [copy](#) operation.
- `VK_ACCESS_HOST_READ_BIT` specifies read access by a host operation. Accesses of this type are not performed through a resource, but directly on memory.
- `VK_ACCESS_HOST_WRITE_BIT` specifies write access by a host operation. Accesses of this type are not performed through a resource, but directly on memory.
- `VK_ACCESS_MEMORY_READ_BIT` specifies all read accesses. It is always valid in any access mask, and is treated as equivalent to setting all `READ` access flags that are valid where it is used.
- `VK_ACCESS_MEMORY_WRITE_BIT` specifies all write accesses. It is always valid in any access mask, and is treated as equivalent to setting all `WRITE` access flags that are valid where it is used.
- `VK_ACCESS_CONDITIONAL_RENDERING_READ_BIT_EXT` specifies read access to a predicate as part of conditional rendering.
- `VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT` specifies write access to a transform feedback buffer made when transform feedback is active.
- `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT` specifies read access to a transform feedback counter buffer which is read when `vkCmdBeginTransformFeedbackEXT` executes.
- `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT` specifies write access to a transform feedback counter buffer which is written when `vkCmdEndTransformFeedbackEXT` executes.
- `VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX` specifies reads from `VkBuffer` inputs to `vkCmdProcessCommandsNVX`.
- `VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX` specifies writes to the target command buffer in `vkCmdProcessCommandsNVX`.
- `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT` is similar to `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`, but also includes [advanced blend operations](#).
- `VK_ACCESS_SHADING_RATE_IMAGE_READ_BIT_NV` specifies read access to a shading rate image as part of a drawing command, as bound by `vkCmdBindShadingRateImageNV`.
- `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NV` specifies read access to an acceleration structure as part of a trace or build command.
- `VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NV` specifies write access to an acceleration structure as part of a build command.
- `VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT` specifies read access to a [fragment density map](#)

## attachment during dynamic fragment density map operations

Certain access types are only performed by a subset of pipeline stages. Any synchronization command that takes both stage masks and access masks uses both to define the [access scopes](#) - only the specified access types performed by the specified stages are included in the access scope. An application **must** not specify an access flag in a synchronization command if it does not include a pipeline stage in the corresponding stage mask that is able to perform accesses of that type. The following table lists, for each access flag, which pipeline stages **can** perform that type of access.

Table 4. Supported access types

Access flag	Supported pipeline stages
<code>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</code>	<code>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</code>
<code>VK_ACCESS_INDEX_READ_BIT</code>	<code>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</code>
<code>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</code>	<code>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</code>
<code>VK_ACCESS_UNIFORM_READ_BIT</code>	<code>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</code> , or <code>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</code>
<code>VK_ACCESS_SHADER_READ_BIT</code>	<code>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</code> , or <code>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</code>
<code>VK_ACCESS_SHADER_WRITE_BIT</code>	<code>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV</code> , <code>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</code> , <code>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</code> , or <code>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</code>
<code>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</code>	<code>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</code>
<code>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</code>	<code>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</code>
<code>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</code>	<code>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</code>
<code>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</code>	<code>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</code> , or <code>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</code>

Access flag	Supported pipeline stages
<code>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</code>	<code>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</code> , or <code>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</code>
<code>VK_ACCESS_TRANSFER_READ_BIT</code>	<code>VK_PIPELINE_STAGE_TRANSFER_BIT</code>
<code>VK_ACCESS_TRANSFER_WRITE_BIT</code>	<code>VK_PIPELINE_STAGE_TRANSFER_BIT</code>
<code>VK_ACCESS_HOST_READ_BIT</code>	<code>VK_PIPELINE_STAGE_HOST_BIT</code>
<code>VK_ACCESS_HOST_WRITE_BIT</code>	<code>VK_PIPELINE_STAGE_HOST_BIT</code>
<code>VK_ACCESS_MEMORY_READ_BIT</code>	Any
<code>VK_ACCESS_MEMORY_WRITE_BIT</code>	Any
<code>VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT</code>	<code>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</code>
<code>VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX</code>	<code>VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX</code>
<code>VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX</code>	<code>VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX</code>
<code>VK_ACCESS_CONDITIONAL_RENDERING_READ_BIT_EXT</code>	<code>VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT</code>
<code>VK_ACCESS_SHADING_RATE_IMAGE_READ_BIT_NV</code>	<code>VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV</code>
<code>VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT</code>	<code>VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT</code>
<code>VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT</code>	<code>VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT</code>
<code>VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT</code>	<code>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</code>
<code>VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NV</code>	<code>VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV</code> , or <code>VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV</code>
<code>VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NV</code>	<code>VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV</code>
<code>VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT</code>	<code>VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT</code>

If a memory object does not have the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` property, then `vkFlushMappedMemoryRanges` **must** be called in order to guarantee that writes to the memory object from the host are made available to the host domain, where they **can** be further made available to the device domain via a domain operation. Similarly, `vkInvalidateMappedMemoryRanges` **must** be called to guarantee that writes which are available to the host domain are made visible to host operations.

If the memory object does have the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` property flag, writes to the memory object from the host are automatically made available to the host domain. Similarly, writes made available to the host domain are automatically made visible to the host.

#### Note

The `vkQueueSubmit` command **automatically performs a domain operation from host to device** for all writes performed before the command executes, so in most cases an explicit memory barrier is not needed for this case. In the few circumstances where a submit does not occur between the host write and the device read access, writes **can** be made available by using an explicit memory barrier.



```
typedef VkFlags VkAccessFlags;
```

`VkAccessFlags` is a bitmask type for setting a mask of zero or more `VkAccessFlagBits`.

### 6.1.4. Framebuffer Region Dependencies

Pipeline stages that operate on, or with respect to, the framebuffer are collectively the *framebuffer-space* pipeline stages. These stages are:

- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`
- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`

For these pipeline stages, an execution or memory dependency from the first set of operations to the second set **can** either be a single *framebuffer-global* dependency, or split into multiple *framebuffer-local* dependencies. A dependency with non-framebuffer-space pipeline stages is neither framebuffer-global nor framebuffer-local.

A *framebuffer region* is a set of sample (x, y, layer, sample) coordinates that is a subset of the entire framebuffer.

Both *synchronization scopes* of a framebuffer-local dependency include only the operations performed within corresponding framebuffer regions (as defined below). No ordering guarantees are made between different framebuffer regions for a framebuffer-local dependency.

Both *synchronization scopes* of a framebuffer-global dependency include operations on all framebuffer-regions.

If the first synchronization scope includes operations on pixels/fragments with N samples and the second synchronization scope includes operations on pixels/fragments with M samples, where N does not equal M, then a framebuffer region containing all samples at a given (x, y, layer) coordinate in the first synchronization scope corresponds to a region containing all samples at the same coordinate in the second synchronization scope. In other words, it is a pixel granularity dependency. If N equals M, then a framebuffer region containing a single (x, y, layer, sample) coordinate in the first synchronization scope corresponds to a region containing the same sample at the same coordinate in the second synchronization scope. In other words, it is a sample granularity dependency.

*Note*



Since fragment invocations are not specified to run in any particular groupings, the size of a framebuffer region is implementation-dependent, not known to the application, and **must** be assumed to be no larger than specified above.

*Note*

Practically, the pixel vs sample granularity dependency means that if an input attachment has a different number of samples than the pipeline's `rasterizationSamples`, then a fragment **can** access any sample in the input attachment's pixel even if it only uses framebuffer-local dependencies. If the input attachment has the same number of samples, then the fragment **can** only access the covered samples in its input `SampleMask` (i.e. the fragment operations happen-after a framebuffer-local dependency for each sample the fragment covers). To access samples that are not covered, a framebuffer-global dependency is required.



If a synchronization command includes a `dependencyFlags` parameter, and specifies the `VK_DEPENDENCY_BY_REGION_BIT` flag, then it defines framebuffer-local dependencies for the framebuffer-space pipeline stages in that synchronization command, for all framebuffer regions. If no `dependencyFlags` parameter is included, or the `VK_DEPENDENCY_BY_REGION_BIT` flag is not specified, then a framebuffer-global dependency is specified for those stages. The `VK_DEPENDENCY_BY_REGION_BIT` flag does not affect the dependencies between non-framebuffer-space pipeline stages, nor does it affect the dependencies between framebuffer-space and non-framebuffer-space pipeline stages.

*Note*

Framebuffer-local dependencies are more optimal for most architectures; particularly tile-based architectures - which can keep framebuffer-regions entirely in on-chip registers and thus avoid external bandwidth across such a dependency. Including a framebuffer-global dependency in your rendering will usually force all implementations to flush data to memory, or to a higher level cache, breaking any potential locality optimizations.



### 6.1.5. View-Local Dependencies

In a render pass instance that has `multiview` enabled, dependencies **can** be either view-local or view-global.

A view-local dependency only includes operations from a single `source view` from the source subpass in the first synchronization scope, and only includes operations from a single `destination view` from the destination subpass in the second synchronization scope. A view-global dependency includes all views in the view mask of the source and destination subpasses in the corresponding synchronization scopes.

If a synchronization command includes a `dependencyFlags` parameter and specifies the `VK_DEPENDENCY_VIEW_LOCAL_BIT` flag, then it defines view-local dependencies for that synchronization command, for all views. If no `dependencyFlags` parameter is included or the `VK_DEPENDENCY_VIEW_LOCAL_BIT` flag is not specified, then a view-global dependency is specified.

### 6.1.6. Device-Local Dependencies

Dependencies **can** be either device-local or non-device-local. A device-local dependency acts as multiple separate dependencies, one for each physical device that executes the synchronization

command, where each dependency only includes operations from that physical device in both synchronization scopes. A non-device-local dependency is a single dependency where both synchronization scopes include operations from all physical devices that participate in the synchronization command. For subpass dependencies, all physical devices in the `VkDeviceGroupRenderPassBeginInfo::deviceMask` participate in the dependency, and for pipeline barriers all physical devices that are set in the command buffer's current device mask participate in the dependency.

If a synchronization command includes a `dependencyFlags` parameter and specifies the `VK_DEPENDENCY_DEVICE_GROUP_BIT` flag, then it defines a non-device-local dependency for that synchronization command. If no `dependencyFlags` parameter is included or the `VK_DEPENDENCY_DEVICE_GROUP_BIT` flag is not specified, then it defines device-local dependencies for that synchronization command, for all participating physical devices.

Semaphore and event dependencies are device-local and only execute on the one physical device that performs the dependency.

## 6.2. Implicit Synchronization Guarantees

A small number of implicit ordering guarantees are provided by Vulkan, ensuring that the order in which commands are submitted is meaningful, and avoiding unnecessary complexity in common operations.

*Submission order* is a fundamental ordering in Vulkan, giving meaning to the order in which [action](#) and [synchronization commands](#) are recorded and submitted to a single queue. Explicit and implicit ordering guarantees between commands in Vulkan all work on the premise that this ordering is meaningful. This order does not itself define any execution or memory dependencies; synchronization commands and other orderings within the API use this ordering to define their scopes.

Submission order for any given set of commands is based on the order in which they were recorded to command buffers and then submitted. This order is determined as follows:

1. The initial order is determined by the order in which `vkQueueSubmit` commands are executed on the host, for a single queue, from first to last.
2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, from lowest index to highest.
3. The order in which command buffers are specified in the `pCommandBuffers` member of `VkSubmitInfo`, from lowest index to highest.
4. The order in which commands were recorded to a command buffer on the host, from first to last:
  - For commands recorded outside a render pass, this includes all other commands recorded outside a render pass, including `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands; it does not directly include commands inside a render pass.
  - For commands recorded inside a render pass, this includes all other commands recorded inside the same subpass, including the `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands that delimit the same render pass instance; it does not include commands

recorded to other subpasses.

Action and synchronization commands recorded to a command buffer execute the `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` pipeline stage in submission order - forming an implicit execution dependency between this stage in each command.

State commands do not execute any operations on the device, instead they set the state of the command buffer when they execute on the host, in the order that they are recorded. Action commands consume the current state of the command buffer when they are recorded, and will execute state changes on the device as required to match the recorded state.

Query commands, the order of primitives passing through the graphics pipeline and image layout transitions as part of an image memory barrier provide additional guarantees based on submission order.

Execution of pipeline stages within a given command also has a loose ordering, dependent only on a single command.

*Signal operation order* is a fundamental ordering in Vulkan, giving meaning to the order in which semaphore and fence signal operations occur when submitted to a single queue. The signal operation order for queue operations is determined as follows:

1. The initial order is determined by the order in which `vkQueuePresentKHR`, `vkQueueSubmit`, and `vkQueueBindSparse` commands are executed on the host, for a single queue, from first to last.
2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, from lowest index to highest.
3. The fence signal operation defined by the `fence` parameter of a `vkQueueSubmit` or `vkQueueBindSparse` command is ordered after all semaphore signal operations defined by that command.

Semaphore signal operations defined by a single `VkSubmitInfo` or `VkBindSparseInfo` structure are unordered with respect to other semaphore signal operations defined within the same `VkSubmitInfo` or `VkBindSparseInfo` structure.

The `vkSignalSemaphoreKHR` command does not execute on a queue but instead performs the signal operation from the host. The semaphore signal operation defined by executing a `vkSignalSemaphoreKHR` command happens-after the `vkSignalSemaphoreKHR` command is invoked and happens-before the command returns.

### Note

When signaling timeline semaphores, it is the responsibility of the client to ensure that they are ordered such that the semaphore value is strictly increasing. Because the first synchronization scope for a semaphore signal operation contains all semaphore signal operations which occur earlier in submission order, all semaphore signal operations contained in any given batch are guaranteed to happen-after all semaphore signal operations contained in any previous batches. However, no ordering guarantee is provided between the semaphore signal operations defined within a single batch. This, combined with the requirement that timeline semaphore values strictly increase, means that it is invalid to signal the same timeline semaphore twice within a single batch.

If a client wishes to ensure that some semaphore signal operation happens-after some other semaphore signal operation, it can submit a separate batch containing only semaphore signal operations, which will happen-after the semaphore signal operations in any earlier batches.

When signaling a semaphore from the host, the only ordering guarantee is that the signal operation happens-after when `vkSignalSemaphoreKHR` is called and happens-before it returns. Therefore, it is invalid to call `vkSignalSemaphoreKHR` while there are any outstanding signal operations on that semaphore from any queue submissions unless those queue submissions have some dependency which ensures that they happen-after the host signal operation. One example of this would be if the pending signal operation is, itself, waiting on the same semaphore at a lower value and the call to `vkSignalSemaphoreKHR` signals that lower value. Furthermore, if there are two or more processes or threads signaling the same timeline semaphore from the host, the client must ensure that the `vkSignalSemaphoreKHR` with the lower semaphore value returns before `vkSignalSemaphoreKHR` is called with the higher value.

## 6.3. Fences

Fences are a synchronization primitive that **can** be used to insert a dependency from a queue to the host. Fences have two states - signaled and unsignaled. A fence **can** be signaled as part of the execution of a `queue submission` command. Fences **can** be unsignaled on the host with `vkResetFences`. Fences **can** be waited on by the host with the `vkWaitForFences` command, and the current state **can** be queried with `vkGetFenceStatus`.

As with most objects in Vulkan, fences are an interface to internal data which is typically opaque to applications. This internal data is referred to as a fence's *payload*.

However, in order to enable communication with agents outside of the current device, it is necessary to be able to export that payload to a commonly understood format, and subsequently import from that format as well.

The internal data of a fence **may** include a reference to any resources and pending work associated with signal or unsignal operations performed on that fence object. Mechanisms to import and export that internal data to and from fences are provided [below](#). These mechanisms indirectly

enable applications to share fence state between two or more fences and other synchronization primitives across process and API boundaries.

Fences are represented by `VkFence` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFence)
```

To create a fence, call:

```
VkResult vkCreateFence(  
    VkDevice device,  
    const VkFenceCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkFence* pFence);
```

- `device` is the logical device that creates the fence.
- `pCreateInfo` is a pointer to a `VkFenceCreateInfo` structure containing information about how the fence is to be created.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pFence` is a pointer to a handle in which the resulting fence object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkFenceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pFence` **must** be a valid pointer to a `VkFence` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkFenceCreateInfo` structure is defined as:

```
typedef struct VkFenceCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkFenceCreateFlags flags;
} VkFenceCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkFenceCreateFlagBits` specifying the initial state and behavior of the fence.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_FENCE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkExportFenceCreateInfo` or `VkExportFenceWin32HandleInfoKHR`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be a valid combination of `VkFenceCreateFlagBits` values

```
typedef enum VkFenceCreateFlagBits {
    VK_FENCE_CREATE_SIGNALED_BIT = 0x00000001,
    VK_FENCE_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkFenceCreateFlagBits;
```

- `VK_FENCE_CREATE_SIGNALED_BIT` specifies that the fence object is created in the signaled state. Otherwise, it is created in the unsignaled state.

```
typedef VkFlags VkFenceCreateFlags;
```

`VkFenceCreateFlags` is a bitmask type for setting a mask of zero or more `VkFenceCreateFlagBits`.

To create a fence whose payload **can** be exported to external handles, add the `VkExportFenceCreateInfo` structure to the `pNext` chain of the `VkFenceCreateInfo` structure. The `VkExportFenceCreateInfo` structure is defined as:

```
typedef struct VkExportFenceCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalFenceHandleTypeFlags handleTypes;
} VkExportFenceCreateInfo;
```

or the equivalent

```
typedef VkExportFenceCreateInfo VkExportFenceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of `VkExternalFenceHandleTypeFlagBits` specifying one or more fence handle types the application **can** export from the resulting fence. The application **can** request multiple handle types for the same fence.

## Valid Usage

- The bits in `handleTypes` must be supported and compatible, as reported by `VkExternalFenceProperties`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO`
- `handleTypes` **must** be a valid combination of `VkExternalFenceHandleTypeFlagBits` values

To specify additional attributes of NT handles exported from a fence, add the `VkExportFenceWin32HandleInfoKHR` structure to the `pNext` chain of the `VkFenceCreateInfo` structure. The `VkExportFenceWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkExportFenceWin32HandleInfoKHR {
    VkStructureType          sType;
    const void*               pNext;
    const SECURITY_ATTRIBUTES* pAttributes;
    DWORD                    dwAccess;
    LPCWSTR                  name;
} VkExportFenceWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pAttributes` is a pointer to a Windows `SECURITY_ATTRIBUTES` structure specifying security attributes of the handle.
- `dwAccess` is a `DWORD` specifying access rights of the handle.
- `name` is a null-terminated UTF-16 string to associate with the underlying synchronization primitive referenced by NT handles exported from the created fence.

If this structure is not present, or if `pAttributes` is set to `NULL`, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in

the MSDN documentation for “Synchronization Object Security and Access Rights”<sup>1</sup>. Further, if the structure is not present, the access rights will be

`DXGI_SHARED_RESOURCE_READ | DXGI_SHARED_RESOURCE_WRITE`

for handles of the following types:

`VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT`

**1**

<https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-object-security-and-access-rights>

## Valid Usage

- If `VkExportFenceCreateInfo::handleTypes` does not include `VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VkExportFenceWin32HandleInfoKHR` **must** not be in the `pNext` chain of `VkFenceCreateInfo`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_FENCE_WIN32_HANDLE_INFO_KHR`
- If `pAttributes` is not `NULL`, `pAttributes` **must** be a valid pointer to a valid `SECURITY_ATTRIBUTES` value

To export a Windows handle representing the state of a fence, call:

```
VkResult vkGetFenceWin32HandleKHR(  
    VkDevice device,  
    const VkFenceGetWin32HandleInfoKHR* pGetWin32HandleInfo,  
    HANDLE* pHandle);
```

- `device` is the logical device that created the fence being exported.
- `pGetWin32HandleInfo` is a pointer to a `VkFenceGetWin32HandleInfoKHR` structure containing parameters of the export operation.
- `pHandle` will return the Windows handle representing the fence state.

For handle types defined as NT handles, the handles returned by `vkGetFenceWin32HandleKHR` are owned by the application. To avoid leaking resources, the application **must** release ownership of them using the `CloseHandle` system call when they are no longer needed.

Exporting a Windows handle from a fence **may** have side effects depending on the transference of the specified handle type, as described in [Importing Fence Payloads](#).

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pGetWin32HandleInfo` **must** be a valid pointer to a valid `VkFenceGetWin32HandleInfoKHR` structure
- `pHandle` **must** be a valid pointer to a `HANDLE` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkFenceGetWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkFenceGetWin32HandleInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkFence                  fence;
    VkExternalFenceHandleTypeFlagBits handleType;
} VkFenceGetWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `fence` is the fence from which state will be exported.
- `handleType` is the type of handle requested.

The properties of the handle returned depend on the value of `handleType`. See `VkExternalFenceHandleTypeFlagBits` for a description of the properties of the defined external fence handle types.

## Valid Usage

- `handleType` **must** have been included in `VkExportFenceCreateInfo::handleTypes` when the `fence`'s current payload was created.
- If `handleType` is defined as an NT handle, `vkGetFenceWin32HandleKHR` **must** be called no more than once for each valid unique combination of `fence` and `handleType`.
- `fence` **must** not currently have its payload replaced by an imported payload as described below in [Importing Fence Payloads](#) unless that imported payload's handle type was included in `VkExternalFenceProperties::exportFromImportedHandleTypes` for `handleType`.
- If `handleType` refers to a handle type with copy payload transference semantics, `fence` **must** be signaled, or have an associated [fence signal operation](#) pending execution.
- `handleType` **must** be defined as an NT handle or a global share handle.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_FENCE_GET_WIN32_HANDLE_INFO_KHR`
- `pNext` **must** be `NULL`
- `fence` **must** be a valid `VkFence` handle
- `handleType` **must** be a valid `VkExternalFenceHandleTypeFlagBits` value

To export a POSIX file descriptor representing the payload of a fence, call:

```
VkResult vkGetFenceFdKHR(  
    VkDevice device,  
    const VkFenceGetFdInfoKHR* pGetFdInfo,  
    int* pFd);
```

- `device` is the logical device that created the fence being exported.
- `pGetFdInfo` is a pointer to a `VkFenceGetFdInfoKHR` structure containing parameters of the export operation.
- `pFd` will return the file descriptor representing the fence payload.

Each call to `vkGetFenceFdKHR` **must** create a new file descriptor and transfer ownership of it to the application. To avoid leaking resources, the application **must** release ownership of the file descriptor when it is no longer needed.

### Note



Ownership can be released in many ways. For example, the application can call `close()` on the file descriptor, or transfer ownership back to Vulkan by using the file descriptor to import a fence payload.

If `pGetFdInfo->handleType` is `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` and the fence is signaled at the time `vkGetFenceFdKHR` is called, `pFd` **may** return the value `-1` instead of a valid file descriptor.

Where supported by the operating system, the implementation **must** set the file descriptor to be closed automatically when an `execve` system call is made.

Exporting a file descriptor from a fence **may** have side effects depending on the transference of the specified handle type, as described in [Importing Fence State](#).

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pGetFdInfo` **must** be a valid pointer to a valid `VkFenceGetFdInfoKHR` structure
- `pFd` **must** be a valid pointer to an `int` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkFenceGetFdInfoKHR` structure is defined as:

```
typedef struct VkFenceGetFdInfoKHR {
    VkStructureType           sType;
    const void*                pNext;
    VkFence                    fence;
    VkExternalFenceHandleTypeFlagBits handleType;
} VkFenceGetFdInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `fence` is the fence from which state will be exported.
- `handleType` is the type of handle requested.

The properties of the file descriptor returned depend on the value of `handleType`. See `VkExternalFenceHandleTypeFlagBits` for a description of the properties of the defined external fence handle types.

## Valid Usage

- `handleType` **must** have been included in `VkExportFenceCreateInfo::handleTypes` when `fence`'s current payload was created.
- If `handleType` refers to a handle type with copy payload transference semantics, `fence` **must** be signaled, or have an associated `fence signal operation` pending execution.
- `fence` **must** not currently have its payload replaced by an imported payload as described below in [Importing Fence Payloads](#) unless that imported payload's handle type was included in `VkExternalFenceProperties::exportFromImportedHandleTypes` for `handleType`.
- `handleType` **must** be defined as a POSIX file descriptor handle.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR`
- `pNext` **must** be `NULL`
- `fence` **must** be a valid `VkFence` handle
- `handleType` **must** be a valid `VkExternalFenceHandleTypeFlagBits` value

To destroy a fence, call:

```
void vkDestroyFence(  
    VkDevice device,  
    VkFence fence,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the fence.
- `fence` is the handle of the fence to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All `queue submission` commands that refer to `fence` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `fence` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `fence` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be a valid `VkFence` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `fence` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `fence` **must** be externally synchronized

To query the status of a fence from the host, call:

```
VkResult vkGetFenceStatus(  
    VkDevice device,  
    VkFence fence);
```

- `device` is the logical device that owns the fence.
- `fence` is the handle of the fence to query.

Upon success, `vkGetFenceStatus` returns the status of the fence object, with the following return codes:

*Table 5. Fence Object Status Codes*

Status	Meaning
<code>VK_SUCCESS</code>	The fence specified by <code>fence</code> is signaled.
<code>VK_NOT_READY</code>	The fence specified by <code>fence</code> is unsignaled.
<code>VK_ERROR_DEVICE_LOST</code>	The device has been lost. See <a href="#">Lost Device</a> .

If a `queue submission` command is pending execution, then the value returned by this command **may** immediately be out of date.

If the device has been lost (see [Lost Device](#)), `vkGetFenceStatus` **may** return any of the above status codes. If the device has been lost and `vkGetFenceStatus` is called repeatedly, it will eventually return either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `fence` **must** be a valid `VkFence` handle
- `fence` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_NOT_READY`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

To set the state of fences to unsignaled from the host, call:

```
VkResult vkResetFences(  
    VkDevice                      device,  
    uint32_t                      fenceCount,  
    const VkFence*                pFences);
```

- `device` is the logical device that owns the fences.
- `fenceCount` is the number of fences to reset.
- `pFences` is a pointer to an array of fence handles to reset.

If any member of `pFences` currently has its [payload imported](#) with temporary permanence, that fence's prior permanent payload is first restored. The remaining operations described therefore operate on the restored payload.

When `vkResetFences` is executed on the host, it defines a *fence unsignal operation* for each fence, which resets the fence to the unsignaled state.

If any member of `pFences` is already in the unsignaled state when `vkResetFences` is executed, then `vkResetFences` has no effect on that fence.

## Valid Usage

- Each element of `pFences` **must** not be currently associated with any queue command that has not yet completed execution on that queue

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pFences` **must** be a valid pointer to an array of `fenceCount` valid `VkFence` handles
- `fenceCount` **must** be greater than `0`
- Each element of `pFences` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to each member of `pFences` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

When a fence is submitted to a queue as part of a [queue submission](#) command, it defines a memory dependency on the batches that were submitted as part of that command, and defines a *fence signal operation* which sets the fence to the signaled state.

The first [synchronization scope](#) includes every batch submitted in the same [queue submission](#) command. Fence signal operations that are defined by `vkQueueSubmit` additionally include in the first synchronization scope all commands that occur earlier in [submission order](#). Fence signal operations that are defined by `vkQueueSubmit` or `vkQueueBindSparse` additionally include in the first synchronization scope any semaphore and fence signal operations that occur earlier in [signal operation order](#).

The second [synchronization scope](#) only includes the fence signal operation.

The first [access scope](#) includes all memory access performed by the device.

The second [access scope](#) is empty.

To wait for one or more fences to enter the signaled state on the host, call:

```
VkResult vkWaitForFences(  
    VkDevice device,  
    uint32_t fenceCount,  
    const VkFence* pFences,  
    VkBool32 waitAll,  
    uint64_t timeout);
```

- `device` is the logical device that owns the fences.
- `fenceCount` is the number of fences to wait on.
- `pFences` is a pointer to an array of `fenceCount` fence handles.
- `waitAll` is the condition that **must** be satisfied to successfully unblock the wait. If `waitAll` is `VK_TRUE`, then the condition is that all fences in `pFences` are signaled. Otherwise, the condition is that at least one fence in `pFences` is signaled.
- `timeout` is the timeout period in units of nanoseconds. `timeout` is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which **may** be substantially longer than one nanosecond, and **may** be longer than the requested period.

If the condition is satisfied when `vkWaitForFences` is called, then `vkWaitForFences` returns immediately. If the condition is not satisfied at the time `vkWaitForFences` is called, then `vkWaitForFences` will block and wait up to `timeout` nanoseconds for the condition to become satisfied.

If `timeout` is zero, then `vkWaitForFences` does not wait, but simply returns the current state of the fences. `VK_TIMEOUT` will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the specified timeout period expires before the condition is satisfied, `vkWaitForFences` returns `VK_TIMEOUT`. If the condition is satisfied before `timeout` nanoseconds has expired, `vkWaitForFences` returns `VK_SUCCESS`.

If device loss occurs (see [Lost Device](#)) before the timeout has expired, `vkWaitForFences` **must** return in finite time with either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`.

*Note*

While we guarantee that `vkWaitForFences` **must** return in finite time, no guarantees are made that it returns immediately upon device loss. However, the client can reasonably expect that the delay will be on the order of seconds and that calling `vkWaitForFences` will not result in a permanently (or seemingly permanently) dead process.



## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pFences` **must** be a valid pointer to an array of `fenceCount` valid `VkFence` handles
- `fenceCount` **must** be greater than `0`
- Each element of `pFences` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_TIMEOUT`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

An execution dependency is defined by waiting for a fence to become signaled, either via `vkWaitForFences` or by polling on `vkGetFenceStatus`.

The first [synchronization scope](#) includes only the fence signal operation.

The second [synchronization scope](#) includes the host operations of `vkWaitForFences` or `vkGetFenceStatus` indicating that the fence has become signaled.

#### Note

Signaling a fence and waiting on the host does not guarantee that the results of memory accesses will be visible to the host, as the access scope of a memory dependency defined by a fence only includes device access. A [memory barrier](#) or other memory dependency **must** be used to guarantee this. See the description of [host access types](#) for more information.



### 6.3.1. Alternate Methods to Signal Fences

Besides submitting a fence to a queue as part of a [queue submission](#) command, a fence **may** also be signaled when a particular event occurs on a device or display.

To create a fence that will be signaled when an event occurs on a device, call:

```

VkResult vkRegisterDeviceEventEXT(
    VkDevice device,
    const VkDeviceEventInfoEXT* pDeviceEventInfo,
    const VkAllocationCallbacks* pAllocator,
    VkFence* pFence);

```

- `device` is a logical device on which the event **may** occur.
- `pDeviceEventInfo` is a pointer to a `VkDeviceEventInfoEXT` structure describing the event of interest to the application.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pFence` is a pointer to a handle in which the resulting fence object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pDeviceEventInfo` **must** be a valid pointer to a valid `VkDeviceEventInfoEXT` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pFence` **must** be a valid pointer to a `VkFence` handle

## Return Codes

### Success

- `VK_SUCCESS`

The `VkDeviceEventInfoEXT` structure is defined as:

```

typedef struct VkDeviceEventInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDeviceEventTypeEXT deviceEvent;
} VkDeviceEventInfoEXT;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `device` is a `VkDeviceEventTypeEXT` value specifying when the fence will be signaled.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT`
- `pNext` **must** be `NULL`
- `deviceEvent` **must** be a valid `VkDeviceEventTypeEXT` value

Possible values of `VkDeviceEventInfoEXT::device`, specifying when a fence will be signaled, are:

```
typedef enum VkDeviceEventTypeEXT {
    VK_DEVICE_EVENT_TYPE_DISPLAY_HOTPLUG_EXT = 0,
    VK_DEVICE_EVENT_TYPE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDeviceEventTypeEXT;
```

- `VK_DEVICE_EVENT_TYPE_DISPLAY_HOTPLUG_EXT` specifies that the fence is signaled when a display is plugged into or unplugged from the specified device. Applications **can** use this notification to determine when they need to re-enumerate the available displays on a device.

To create a fence that will be signaled when an event occurs on a `VkDisplayKHR` object, call:

```
VkResult vkRegisterDisplayEventEXT(
    VkDevice                                     device,
    VkDisplayKHR                                display,
    const VkDisplayEventCreateInfo*               pDisplayCreateInfo,
    const VkAllocationCallbacks*                 pAllocator,
    VkFence*                                    pFence);
```

- `device` is a logical device associated with `display`
- `display` is the display on which the event **may** occur.
- `pDisplayCreateInfo` is a pointer to a `VkDisplayEventCreateInfoEXT` structure describing the event of interest to the application.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pFence` is a pointer to a handle in which the resulting fence object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `display` **must** be a valid `VkDisplayKHR` handle
- `pDisplayCreateInfo` **must** be a valid pointer to a valid `VkDisplayEventCreateInfoEXT` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pFence` **must** be a valid pointer to a `VkFence` handle

## Return Codes

### Success

- `VK_SUCCESS`

The `VkDisplayEventInfoEXT` structure is defined as:

```
typedef struct VkDisplayEventInfoEXT {
    VkStructureType         sType;
    const void*             pNext;
    VkDisplayEventTypeEXT   displayEvent;
} VkDisplayEventInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `displayEvent` is a `VkDisplayEventTypeEXT` specifying when the fence will be signaled.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT`
- `pNext` **must** be `NULL`
- `displayEvent` **must** be a valid `VkDisplayEventTypeEXT` value

Possible values of `VkDisplayEventInfoEXT::displayEvent`, specifying when a fence will be signaled, are:

```
typedef enum VkDisplayEventTypeEXT {
    VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT = 0,
    VK_DISPLAY_EVENT_TYPE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDisplayEventTypeEXT;
```

- `VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT` specifies that the fence is signaled when the first pixel of the next display refresh cycle leaves the display engine for the display.

### 6.3.2. Importing Fence Payloads

Applications **can** import a fence payload into an existing fence using an external fence handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the fence will be *restored* to its permanent state the next time that fence is passed to `vkResetFences`.

*Note*



Restoring a fence to its prior permanent payload is a distinct operation from resetting a fence payload. See [vkResetFences](#) for more detail.

Performing a subsequent temporary import on a fence before resetting it has no effect on this requirement; the next unsignal of the fence **must** still restore its last permanent state. A permanent payload import behaves as if the target fence was destroyed, and a new fence was created with the same handle but the imported payload. Because importing a fence payload temporarily or permanently detaches the existing payload from a fence, similar usage restrictions to those applied to [vkDestroyFence](#) are applied to any command that imports a fence payload. Which of these import types is used is referred to as the import operation's *permanence*. Each handle type supports either one or both types of permanence.

The implementation **must** perform the import operation by either referencing or copying the payload referred to by the specified external fence handle, depending on the handle's type. The import method used is referred to as the handle type's *transference*. When using handle types with reference transference, importing a payload to a fence adds the fence to the set of all fences sharing that payload. This set includes the fence from which the payload was exported. Fence signaling, waiting, and resetting operations performed on any fence in the set **must** behave as if the set were a single fence. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Fence signaling, waiting, and resetting operations performed on the target of copy imports **must** not affect any other fence or payload.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a fence payload to a handle with copy transference has the same side effects on the source fence's payload as executing a fence reset operation. If the fence was using a temporarily imported payload, the fence's prior permanent payload will be restored.

*Note*



The tables [Handle Types Supported by VkImportFenceWin32HandleInfoKHR](#) and [Handle Types Supported by VkImportFenceFdInfoKHR](#) define the permanence and transference of each handle type.

[External synchronization](#) allows implementations to modify an object's internal state, i.e. payload, without internal synchronization. However, for fences sharing a payload across processes, satisfying the external synchronization requirements of [VkFence](#) parameters as if all fences in the set were the same object is sometimes infeasible. Satisfying valid usage constraints on the state of a fence would similarly require impractical coordination or levels of trust between processes. Therefore, these constraints only apply to a specific fence handle, not to its payload. For distinct fence objects which share a payload:

- If multiple commands which queue a signal operation, or which unsignal a fence, are called concurrently, behavior will be as if the commands were called in an arbitrary sequential order.
- If a queue submission command is called with a fence that is sharing a payload, and the payload is already associated with another queue command that has not yet completed execution, either one or both of the commands will cause the fence to become signaled when they complete

execution.

- If a fence payload is reset while it is associated with a queue command that has not yet completed execution, the payload will become unsignaled, but **may** become signaled again when the command completes execution.
- In the preceding cases, any of the devices associated with the fences sharing the payload **may** be lost, or any of the queue submission or fence reset commands **may** return `VK_ERROR_INITIALIZATION_FAILED`.

Other than these non-deterministic results, behavior is well defined. In particular:

- The implementation **must** not crash or enter an internally inconsistent state where future valid Vulkan commands might cause undefined results,
- Timeouts on future wait commands on fences sharing the payload **must** be effective.

*Note*

These rules allow processes to synchronize access to shared memory without trusting each other. However, such processes must still be cautious not to use the shared fence for more than synchronizing access to the shared memory. For example, a process should not use a fence with shared payload to tell when commands it submitted to a queue have completed and objects used by those commands may be destroyed, since the other process can accidentally or maliciously cause the fence to signal before the commands actually complete.



When a fence is using an imported payload, its `VkExportFenceCreateInfo::handleTypes` value is that specified when creating the fence from which the payload was exported, rather than that specified when creating the fence. Additionally, `VkExternalFenceProperties::exportFromImportedHandleTypes` restricts which handle types **can** be exported from such a fence based on the specific handle type used to import the current payload. Passing a fence to `vkAcquireNextImageKHR` is equivalent to temporarily importing a fence payload to that fence.

*Note*

Because the exportable handle types of an imported fence correspond to its current imported payload, and `vkAcquireNextImageKHR` behaves the same as a temporary import operation for which the source fence is opaque to the application, applications have no way of determining whether any external handle types **can** be exported from a fence in this state. Therefore, applications **must** not attempt to export handles from fences using a temporarily imported payload from `vkAcquireNextImageKHR`.



When importing a fence payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations **must** perform sufficient validation of external handles to ensure that the operation results in a valid fence which will not cause program termination, device loss, queue stalls, host thread stalls, or corruption of other resources when used as allowed according to its import parameters. If the external handle provided does not meet these requirements, the implementation **must** fail the fence payload import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

To import a fence payload from a Windows handle, call:

```
VkResult vkImportFenceWin32HandleKHR(  
    VkDevice device,  
    const VkImportFenceWin32HandleInfoKHR* pImportFenceWin32HandleInfo);
```

- `device` is the logical device that created the fence.
- `pImportFenceWin32HandleInfo` is a pointer to a `VkImportFenceWin32HandleInfoKHR` structure specifying the fence and import parameters.

Importing a fence payload from Windows handles does not transfer ownership of the handle to the Vulkan implementation. For handle types defined as NT handles, the application **must** release ownership using the `CloseHandle` system call when the handle is no longer needed.

Applications **can** import the same fence payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pImportFenceWin32HandleInfo` **must** be a valid pointer to a `VkImportFenceWin32HandleInfoKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkImportFenceWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkImportFenceWin32HandleInfoKHR {  
    VkStructureType sType;  
    const void* pNext;  
    VkFence fence;  
    VkFenceImportFlags flags;  
    VkExternalFenceHandleTypeFlagBits handleType;  
    HANDLE handle;  
    LPCWSTR name;  
} VkImportFenceWin32HandleInfoKHR;
```

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `fence` is the fence into which the state will be imported.
- `flags` is a bitmask of `VkFenceImportFlagBits` specifying additional parameters for the fence payload import operation.
- `handleType` specifies the type of `handle`.
- `handle` is the external handle to import, or `NULL`.
- `name` is a null-terminated UTF-16 string naming the underlying synchronization primitive to import, or `NULL`.

The handle types supported by `handleType` are:

*Table 6. Handle Types Supported by `VkImportFenceWin32HandleInfoKHR`*

Handle Type	Transference	Permanence Supported
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT</code>	Reference	Temporary,Permanent
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT</code>	Reference	Temporary,Permanent

## Valid Usage

- `handleType` **must** be a value included in the Handle Types Supported by `VkImportFenceWin32HandleInfoKHR` table.
- If `handleType` is not `VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `name` **must** be `NULL`.
- If `handleType` is not `0` and `handle` is `NULL`, `name` **must** name a valid synchronization primitive of the type specified by `handleType`.
- If `handleType` is not `0` and `name` is `NULL`, `handle` **must** be a valid handle of the type specified by `handleType`.
- If `handle` is not `NULL`, `name` must be `NULL`.
- If `handle` is not `NULL`, it **must** obey any requirements listed for `handleType` in [external fence handle types compatibility](#).
- If `name` is not `NULL`, it **must** obey any requirements listed for `handleType` in [external fence handle types compatibility](#).

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_FENCE_WIN32_HANDLE_INFO_KHR`
- `pNext` **must** be `NULL`
- `fence` **must** be a valid `VkFence` handle
- `flags` **must** be a valid combination of `VkFenceImportFlagBits` values
- If `handleType` is not `0`, `handleType` **must** be a valid `VkExternalFenceHandleTypeFlagBits` value

## Host Synchronization

- Host access to `fence` **must** be externally synchronized

To import a fence payload from a POSIX file descriptor, call:

```
VkResult vkImportFenceFdKHR(  
    VkDevice device,  
    const VkImportFenceFdInfoKHR* pImportFenceFdInfo);
```

- `device` is the logical device that created the fence.
- `pImportFenceFdInfo` is a pointer to a `VkImportFenceFdInfoKHR` structure specifying the fence and import parameters.

Importing a fence payload from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application **must** not perform any operations on the file descriptor after a successful import.

Applications **can** import the same fence payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

## Valid Usage

- `fence` **must** not be associated with any queue command that has not yet completed execution on that queue

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pImportFenceFdInfo` **must** be a valid pointer to a valid `VkImportFenceFdInfoKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkImportFenceFdInfoKHR` structure is defined as:

```
typedef struct VkImportFenceFdInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkFence                  fence;
    VkFenceImportFlags        flags;
    VkExternalFenceHandleTypeFlagBits handleType;
    int                      fd;
} VkImportFenceFdInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `fence` is the fence into which the payload will be imported.
- `flags` is a bitmask of `VkFenceImportFlagBits` specifying additional parameters for the fence payload import operation.
- `handleType` specifies the type of `fd`.
- `fd` is the external handle to import.

The handle types supported by `handleType` are:

Table 7. Handle Types Supported by `VkImportFenceFdInfoKHR`

Handle Type	Transference	Permanence Supported
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT</code>	Reference	Temporary,Permanent
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT</code>	Copy	Temporary

## Valid Usage

- `handleType` **must** be a value included in the Handle Types Supported by `VkImportFenceFdInfoKHR` table.
- `fd` **must** obey any requirements listed for `handleType` in external fence handle types compatibility.

If `handleType` is `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT`, the special value `-1` for `fd` is treated like a valid sync file descriptor referring to an object that has already signaled. The import operation will succeed and the `VkFence` will have a temporarily imported payload as if a valid file descriptor had been provided.

*Note*

This special behavior for importing an invalid sync file descriptor allows easier interoperability with other system APIs which use the convention that an invalid sync file descriptor represents work that has already completed and does not need to be waited for. It is consistent with the option for implementations to return a `-1` file descriptor when exporting a `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` from a `VkFence` which is signaled.



## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR`
- `pNext` **must** be `NULL`
- `fence` **must** be a valid `VkFence` handle
- `flags` **must** be a valid combination of `VkFenceImportFlagBits` values
- `handleType` **must** be a valid `VkExternalFenceHandleTypeFlagBits` value

## Host Synchronization

- Host access to `fence` **must** be externally synchronized

Bits which can be set in `VkImportFenceWin32HandleInfoKHR::flags` and `VkImportFenceFdInfoKHR::flags` specifying additional parameters of a fence import operation are:

```
typedef enum VkFenceImportFlagBits {
    VK_FENCE_IMPORT_TEMPORARY_BIT = 0x00000001,
    VK_FENCE_IMPORT_TEMPORARY_BIT_KHR = VK_FENCE_IMPORT_TEMPORARY_BIT,
    VK_FENCE_IMPORT_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkFenceImportFlagBits;
```

or the equivalent

```
typedef VkFenceImportFlagBits VkFenceImportFlagBitsKHR;
```

- `VK_FENCE_IMPORT_TEMPORARY_BIT` specifies that the fence payload will be imported only temporarily, as described in [Importing Fence Payloads](#), regardless of the permanence of `handleType`.

```
typedef VkFlags VkFenceImportFlags;
```

or the equivalent

```
typedef VkFenceImportFlags VkFenceImportFlagsKHR;
```

[VkFenceImportFlags](#) is a bitmask type for setting a mask of zero or more [VkFenceImportFlagBits](#).

## 6.4. Semaphores

Semaphores are a synchronization primitive that **can** be used to insert a dependency between queue operations or between a queue operation and the host. [Binary semaphores](#) have two states - signaled and unsignaled. [Timeline semaphores](#) have a monotonically increasing 64-bit unsigned integer payload and are signaled with respect to a particular reference value. A semaphore **can** be signaled after execution of a queue operation is completed, and a queue operation **can** wait for a semaphore to become signaled before it begins execution. A timeline semaphore **can** additionally be signaled from the host with the [vkSignalSemaphoreKHR](#) command and waited on from the host with the [vkWaitSemaphoresKHR](#) command.

As with most objects in Vulkan, semaphores are an interface to internal data which is typically opaque to applications. This internal data is referred to as a semaphore's *payload*.

However, in order to enable communication with agents outside of the current device, it is necessary to be able to export that payload to a commonly understood format, and subsequently import from that format as well.

The internal data of a semaphore **may** include a reference to any resources and pending work associated with signal or unsignal operations performed on that semaphore object. Mechanisms to import and export that internal data to and from semaphores are provided [below](#). These mechanisms indirectly enable applications to share semaphore state between two or more semaphores and other synchronization primitives across process and API boundaries.

Semaphores are represented by [VkSemaphore](#) handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSemaphore)
```

To create a semaphore, call:

```
VkResult vkCreateSemaphore(  
    VkDevice                                     device,  
    const VkSemaphoreCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSemaphore* pSemaphore);
```

- **device** is the logical device that creates the semaphore.

- `pCreateInfo` is a pointer to a `VkSemaphoreCreateInfo` structure containing information about how the semaphore is to be created.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pSemaphore` is a pointer to a handle in which the resulting semaphore object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkSemaphoreCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSemaphore` **must** be a valid pointer to a `VkSemaphore` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkSemaphoreCreateInfo` structure is defined as:

```
typedef struct VkSemaphoreCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkSemaphoreCreateFlags     flags;
} VkSemaphoreCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkExportSemaphoreCreateInfo`, `VkExportSemaphoreWin32HandleInfoKHR`, or `VkSemaphoreTypeCreateInfoKHR`
- Each `sType` member in the `pNext` chain must be unique
- `flags` must be `0`

```
typedef VkFlags VkSemaphoreCreateFlags;
```

`VkSemaphoreCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To create a semaphore of a specific type, add the `VkSemaphoreTypeCreateInfoKHR` structure to the `pNext` chain of the `VkSemaphoreCreateInfo` structure. The `VkSemaphoreTypeCreateInfoKHR` structure is defined as:

```
typedef struct VkSemaphoreTypeCreateInfoKHR {
    VkStructureType      sType;
    const void*          pNext;
    VkSemaphoreTypeKHR   semaphoreType;
    uint64_t              initialValue;
} VkSemaphoreTypeCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `semaphoreType` is a `VkSemaphoreTypeKHR` value specifying the type of the semaphore.
- `initialValue` is the initial payload value if `semaphoreType` is `VK_SEMAPHORE_TYPE_TIMELINE_KHR`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO_KHR`
- `semaphoreType` must be a valid `VkSemaphoreTypeKHR` value

## Valid Usage

- If the `timelineSemaphore` feature is not enabled, `semaphoreType` must not equal `VK_SEMAPHORE_TYPE_TIMELINE_KHR`
- If `semaphoreType` is `VK_SEMAPHORE_TYPE_BINARY_KHR`, `initialValue` must be zero.

If no instance of the `VkSemaphoreTypeCreateInfoKHR` structure is present in the `pNext` chain of `VkSemaphoreCreateInfo` then the created semaphore will have a default `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR`.

Possible values of `VkSemaphoreTypeCreateInfoKHR::semaphoreType`, specifying the type of a semaphore, are:

```
typedef enum VkSemaphoreTypeKHR {
    VK_SEMAPHORE_TYPE_BINARY_KHR = 0,
    VK_SEMAPHORE_TYPE_TIMELINE_KHR = 1,
    VK_SEMAPHORE_TYPE_MAX_ENUM_KHR = 0x7FFFFFFF
} VkSemaphoreTypeKHR;
```

- `VK_SEMAPHORE_TYPE_BINARY_KHR` specifies a *binary semaphore* type that has a boolean payload indicating whether the semaphore is currently signaled or unsignaled. When created, the semaphore is in the unsignaled state.
- `VK_SEMAPHORE_TYPE_TIMELINE_KHR` specifies a *timeline semaphore* type that has a monotonically increasing 64-bit unsigned integer payload indicating whether the semaphore is signaled with respect to a particular reference value. When created, the semaphore payload has the value given by the `initialValue` field of `VkSemaphoreTypeCreateInfoKHR`.

To create a semaphore whose payload **can** be exported to external handles, add the `VkExportSemaphoreCreateInfo` structure to the `pNext` chain of the `VkSemaphoreCreateInfo` structure. The `VkExportSemaphoreCreateInfo` structure is defined as:

```
typedef struct VkExportSemaphoreCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkExternalSemaphoreHandleTypeFlags handleTypes;
} VkExportSemaphoreCreateInfo;
```

or the equivalent

```
typedef VkExportSemaphoreCreateInfo VkExportSemaphoreCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of `VkExternalSemaphoreHandleTypeFlagBits` specifying one or more semaphore handle types the application **can** export from the resulting semaphore. The application **can** request multiple handle types for the same semaphore.

## Valid Usage

- The bits in `handleTypes` **must** be supported and compatible, as reported by `VkExternalSemaphoreProperties`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO`
- `handleTypes` **must** be a valid combination of `VkExternalSemaphoreHandleTypeFlagBits` values

To specify additional attributes of NT handles exported from a semaphore, add the `VkExportSemaphoreWin32HandleInfoKHR` structure to the `pNext` chain of the `VkSemaphoreCreateInfo` structure. The `VkExportSemaphoreWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkExportSemaphoreWin32HandleInfoKHR {
    VkStructureType          sType;
    const void*               pNext;
    const SECURITY_ATTRIBUTES* pAttributes;
    DWORD                    dwAccess;
    LPCWSTR                  name;
} VkExportSemaphoreWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pAttributes` is a pointer to a Windows `SECURITY_ATTRIBUTES` structure specifying security attributes of the handle.
- `dwAccess` is a `DWORD` specifying access rights of the handle.
- `name` is a null-terminated UTF-16 string to associate with the underlying synchronization primitive referenced by NT handles exported from the created semaphore.

If this structure is not present, or if `pAttributes` is set to `NULL`, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in the MSDN documentation for “Synchronization Object Security and Access Rights”<sup>1</sup>.

For handles of the following types:

`VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT`

The implementation **must** ensure the access rights allow both signal and wait operations on the semaphore.

For handles of the following types:

`VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT`

The access rights **must** be:

**GENERIC\_ALL**

**1**

<https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-object-security-and-access-rights>

## Valid Usage

- If `VkExportSemaphoreCreateInfo::handleTypes` does not include `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT` or `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT`, `VkExportSemaphoreWin32HandleInfoKHR` **must** not be in the `pNext` chain of `VkSemaphoreCreateInfo`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR`
- If `pAttributes` is not `NULL`, `pAttributes` **must** be a valid pointer to a valid `SECURITY_ATTRIBUTES` value

To export a Windows handle representing the payload of a semaphore, call:

```
VkResult vkGetSemaphoreWin32HandleKHR(  
    VkDevice device,  
    const VkSemaphoreGetWin32HandleInfoKHR* pGetWin32HandleInfo,  
    HANDLE* pHandle);
```

- `device` is the logical device that created the semaphore being exported.
- `pGetWin32HandleInfo` is a pointer to a `VkSemaphoreGetWin32HandleInfoKHR` structure containing parameters of the export operation.
- `pHandle` will return the Windows handle representing the semaphore state.

For handle types defined as NT handles, the handles returned by `vkGetSemaphoreWin32HandleKHR` are owned by the application. To avoid leaking resources, the application **must** release ownership of them using the `CloseHandle` system call when they are no longer needed.

Exporting a Windows handle from a semaphore **may** have side effects depending on the transference of the specified handle type, as described in [Importing Semaphore Payloads](#).

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pGetWin32HandleInfo` **must** be a valid pointer to a valid `VkSemaphoreGetWin32HandleInfoKHR` structure
- `pHandle` **must** be a valid pointer to a `HANDLE` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkSemaphoreGetWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkSemaphoreGetWin32HandleInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkSemaphore               semaphore;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkSemaphoreGetWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `semaphore` is the semaphore from which state will be exported.
- `handleType` is the type of handle requested.

The properties of the handle returned depend on the value of `handleType`. See `VkExternalSemaphoreHandleTypeFlagBits` for a description of the properties of the defined external semaphore handle types.

## Valid Usage

- `handleType` **must** have been included in `VkExportSemaphoreCreateInfo::handleTypes` when the `semaphore`'s current payload was created.
- If `handleType` is defined as an NT handle, `vkGetSemaphoreWin32HandleKHR` **must** be called no more than once for each valid unique combination of `semaphore` and `handleType`.
- `semaphore` **must** not currently have its payload replaced by an imported payload as described below in [Importing Semaphore Payloads](#) unless that imported payload's handle type was included in `VkExternalSemaphoreProperties::exportFromImportedHandleTypes` for `handleType`.
- If `handleType` refers to a handle type with copy payload transference semantics, as defined below in [Importing Semaphore Payloads](#), there **must** be no queue waiting on `semaphore`.
- If `handleType` refers to a handle type with copy payload transference semantics, `semaphore` **must** be signaled, or have an associated `semaphore signal operation` pending execution.
- `handleType` **must** be defined as an NT handle or a global share handle.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_KHR`
- `pNext` **must** be `NULL`
- `semaphore` **must** be a valid `VkSemaphore` handle
- `handleType` **must** be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

To export a POSIX file descriptor representing the payload of a semaphore, call:

```
VkResult vkGetSemaphoreFdKHR(  
    VkDevice device,  
    const VkSemaphoreGetFdInfoKHR* pGetFdInfo,  
    int* pFd);
```

- `device` is the logical device that created the semaphore being exported.
- `pGetFdInfo` is a pointer to a `VkSemaphoreGetFdInfoKHR` structure containing parameters of the export operation.
- `pFd` will return the file descriptor representing the semaphore payload.

Each call to `vkGetSemaphoreFdKHR` **must** create a new file descriptor and transfer ownership of it to the application. To avoid leaking resources, the application **must** release ownership of the file descriptor when it is no longer needed.

### Note



Ownership can be released in many ways. For example, the application can call `close()` on the file descriptor, or transfer ownership back to Vulkan by using the file descriptor to import a semaphore payload.

Where supported by the operating system, the implementation **must** set the file descriptor to be closed automatically when an `execve` system call is made.

Exporting a file descriptor from a semaphore **may** have side effects depending on the transference of the specified handle type, as described in [Importing Semaphore State](#).

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pGetFdInfo` **must** be a valid pointer to a valid `VkSemaphoreGetFdInfoKHR` structure
- `pFd` **must** be a valid pointer to an `int` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkSemaphoreGetFdInfoKHR` structure is defined as:

```
typedef struct VkSemaphoreGetFdInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkSemaphore               semaphore;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkSemaphoreGetFdInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `semaphore` is the semaphore from which state will be exported.
- `handleType` is the type of handle requested.

The properties of the file descriptor returned depend on the value of `handleType`. See `VkExternalSemaphoreHandleTypeFlagBits` for a description of the properties of the defined external semaphore handle types.

## Valid Usage

- `handleType` **must** have been included in `VkExportSemaphoreCreateInfo::handleTypes` when `semaphore`'s current payload was created.
- `semaphore` **must** not currently have its payload replaced by an imported payload as described below in [Importing Semaphore Payloads](#) unless that imported payload's handle type was included in `VkExternalSemaphoreProperties::exportFromImportedHandleTypes` for `handleType`.
- If `handleType` refers to a handle type with copy payload transference semantics, as defined below in [Importing Semaphore Payloads](#), there **must** be no queue waiting on `semaphore`.
- If `handleType` refers to a handle type with copy payload transference semantics, `semaphore` **must** be signaled, or have an associated `semaphore signal operation` pending execution.
- `handleType` **must** be defined as a POSIX file descriptor handle.
- If `handleType` refers to a handle type with copy payload transference semantics, `semaphore` **must** have been created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR`.
- If `handleType` refers to a handle type with copy payload transference semantics, `semaphore` **must** have an associated semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) **must** have also been submitted for execution.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR`
- `pNext` **must** be `NULL`
- `semaphore` **must** be a valid `VkSemaphore` handle
- `handleType` **must** be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

To destroy a semaphore, call:

```
void vkDestroySemaphore(  
    VkDevice device,  
    VkSemaphore semaphore,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the semaphore.
- `semaphore` is the handle of the semaphore to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted batches that refer to `semaphore` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `semaphore` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `semaphore` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `semaphore` is not `VK_NULL_HANDLE`, `semaphore` **must** be a valid `VkSemaphore` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `semaphore` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `semaphore` **must** be externally synchronized

### 6.4.1. Semaphore Signaling

When a batch is submitted to a queue via a `queue submission`, and it includes semaphores to be signaled, it defines a memory dependency on the batch, and defines *semaphore signal operations* which set the semaphores to the signaled state.

In case of semaphores created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` the semaphore is considered signaled with respect to the counter value set to be signaled as specified in `VkTimelineSemaphoreSubmitInfoKHR` or `VkSemaphoreSignalInfoKHR`.

The first `synchronization scope` includes every command submitted in the same batch. Semaphore signal operations that are defined by `vkQueueSubmit` additionally include all commands that occur earlier in `submission order`. Semaphore signal operations that are defined by `vkQueueSubmit` or `vkQueueBindSparse` additionally include in the first synchronization scope any semaphore and fence signal operations that occur earlier in `signal operation order`.

The second `synchronization scope` includes only the semaphore signal operation.

The first `access scope` includes all memory access performed by the device.

The second `access scope` is empty.

## 6.4.2. Semaphore Waiting

When a batch is submitted to a queue via a [queue submission](#), and it includes semaphores to be waited on, it defines a memory dependency between prior semaphore signal operations and the batch, and defines *semaphore wait operations*.

Such semaphore wait operations set the semaphores created with a [VkSemaphoreTypeKHR](#) of [VK\\_SEMAPHORE\\_TYPE\\_BINARY\\_KHR](#) to the unsignaled state. In case of semaphores created with a [VkSemaphoreTypeKHR](#) of [VK\\_SEMAPHORE\\_TYPE\\_TIMELINE\\_KHR](#) a prior semaphore signal operation defines a memory dependency with a semaphore wait operation if the value the semaphore is signaled with is greater than or equal to the value the semaphore is waited with, thus the semaphore will continue to be considered signaled with respect to the counter value waited on as specified in [VkTimelineSemaphoreSubmitInfoKHR](#).

The first synchronization scope includes all semaphore signal operations that operate on semaphores waited on in the same batch, and that happen-before the wait completes.

The second [synchronization scope](#) includes every command submitted in the same batch. In the case of [vkQueueSubmit](#), the second synchronization scope is limited to operations on the pipeline stages determined by the [destination stage mask](#) specified by the corresponding element of [pWaitDstStageMask](#). Also, in the case of [vkQueueSubmit](#), the second synchronization scope additionally includes all commands that occur later in [submission order](#).

The first [access scope](#) is empty.

The second [access scope](#) includes all memory access performed by the device.

The semaphore wait operation happens-after the first set of operations in the execution dependency, and happens-before the second set of operations in the execution dependency.

*Note*



Unlike timeline semaphores, fences or events, the act of waiting for a binary semaphore also unsignals that semaphore. Applications **must** ensure that between two such wait operations, the semaphore is signaled again, with execution dependencies used to ensure these occur in order. Binary semaphore waits and signals should thus occur in discrete 1:1 pairs.

#### Note

A common scenario for using `pWaitDstStageMask` with values other than `VK_PIPELINE_STAGE_ALL_COMMANDS_BIT` is when synchronizing a window system presentation operation against subsequent command buffers which render the next frame. In this case, a presentation image **must** not be overwritten until the presentation operation completes, but other pipeline stages **can** execute without waiting. A mask of `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` prevents subsequent color attachment writes from executing until the semaphore signals. Some implementations **may** be able to execute transfer operations and/or vertex processing work before the semaphore is signaled.

If an image layout transition needs to be performed on a presentable image before it is used in a framebuffer, that **can** be performed as the first operation submitted to the queue after acquiring the image, and **should** not prevent other work from overlapping with the presentation operation. For example, a `VkImageMemoryBarrier` could use:



- `srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`
- `srcAccessMask = 0`
- `dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`
- `dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT.`
- `oldLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`
- `newLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`

Alternatively, `oldLayout` **can** be `VK_IMAGE_LAYOUT_UNDEFINED`, if the image's contents need not be preserved.

This barrier accomplishes a dependency chain between previous presentation operations and subsequent color attachment output operations, with the layout transition performed in between, and does not introduce a dependency between previous work and any vertex processing stages. More precisely, the semaphore signals after the presentation operation completes, the semaphore wait stalls the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` stage, and there is a dependency from that same stage to itself with the layout transition performed in between.

### 6.4.3. Semaphore State Requirements For Wait Operations

Before waiting on a semaphore, the application **must** ensure the semaphore is in a valid state for a wait operation. Specifically, when a `semaphore wait operation` is submitted to a queue:

- A binary semaphore **must** be signaled, or have an associated `semaphore signal operation` that is pending execution.
- Any `semaphore signal operations` on which the pending binary semaphore signal operation depends **must** also be completed or pending execution.
- There **must** be no other queue waiting on the same binary semaphore when the operation

executes.

#### 6.4.4. Host Operations on Semaphores

In addition to [semaphore signal operations](#) and [semaphore wait operations](#) submitted to device queues, timeline semaphores support the following host operations:

- Query the current counter value of the semaphore using the [vkGetSemaphoreCounterValueKHR](#) command.
- Wait for a set of semaphores to reach particular counter values using the [vkWaitSemaphoresKHR](#) command.
- Signal the semaphore with a particular counter value from the host using the [vkSignalSemaphoreKHR](#) command.

To query the current counter value of a semaphore created with a [VkSemaphoreTypeKHR](#) of [VK\\_SEMAPHORE\\_TYPE\\_TIMELINE\\_KHR](#) from the host, call:

```
VkResult vkGetSemaphoreCounterValueKHR(  
    VkDevice device,  
    VkSemaphore semaphore,  
    uint64_t* pValue);
```

- **device** is the logical device that owns the semaphore.
- **semaphore** is the handle of the semaphore to query.
- **pValue** is a pointer to a 64-bit integer value in which the current counter value of the semaphore is returned.

*Note*



If a [queue submission](#) command is pending execution, then the value returned by this command **may** immediately be out of date.

#### Valid Usage

- **semaphore must** have been created with a [VkSemaphoreTypeKHR](#) of [VK\\_SEMAPHORE\\_TYPE\\_TIMELINE\\_KHR](#)

#### Valid Usage (Implicit)

- **device must** be a valid [VkDevice](#) handle
- **semaphore must** be a valid [VkSemaphore](#) handle
- **pValue must** be a valid pointer to a [uint64\\_t](#) value
- **semaphore must** have been created, allocated, or retrieved from **device**

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

To wait for a set of semaphores created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` to reach particular counter values on the host, call:

```
VkResult vkWaitSemaphoresKHR(  
    VkDevice device,  
    const VkSemaphoreWaitInfoKHR* pWaitInfo,  
    uint64_t timeout);
```

- `device` is the logical device that owns the semaphore.
- `pWaitInfo` is a pointer to an instance of the `VkSemaphoreWaitInfoKHR` structure containing information about the wait condition.
- `timeout` is the timeout period in units of nanoseconds. `timeout` is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which **may** be substantially longer than one nanosecond, and **may** be longer than the requested period.

If the condition is satisfied when `vkWaitSemaphoresKHR` is called, then `vkWaitSemaphoresKHR` returns immediately. If the condition is not satisfied at the time `vkWaitSemaphoresKHR` is called, then `vkWaitSemaphoresKHR` will block and wait up to `timeout` nanoseconds for the condition to become satisfied.

If `timeout` is zero, then `vkWaitSemaphoresKHR` does not wait, but simply returns information about the current state of the semaphore. `VK_TIMEOUT` will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the specified timeout period expires before the condition is satisfied, `vkWaitSemaphoresKHR` returns `VK_TIMEOUT`. If the condition is satisfied before `timeout` nanoseconds has expired, `vkWaitSemaphoresKHR` returns `VK_SUCCESS`.

If device loss occurs (see [Lost Device](#)) before the timeout has expired, `vkWaitSemaphoresKHR` **must** return in finite time with either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pWaitInfo` **must** be a valid pointer to a valid `VkSemaphoreWaitInfoKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_TIMEOUT`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

The `VkSemaphoreWaitInfoKHR` structure is defined as:

```
typedef struct VkSemaphoreWaitInfoKHR {
    VkStructureType          sType;
    const void*               pNext;
    VkSemaphoreWaitFlagsKHR   flags;
    uint32_t                  semaphoreCount;
    const VkSemaphore*        pSemaphores;
    const uint64_t*           pValues;
} VkSemaphoreWaitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkSemaphoreWaitFlagBitsKHR` specifying additional parameters for the semaphore wait operation.
- `semaphoreCount` is the number of semaphores to wait on.
- `pSemaphores` is a pointer to an array of `semaphoreCount` semaphore handles to wait on.
- `pValues` is a pointer to an array of `semaphoreCount` timeline semaphore values.

## Valid Usage

- All of the elements of `pSemaphores` **must** reference a semaphore that was created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of `VkSemaphoreWaitFlagBitsKHR` values
- `pSemaphores` **must** be a valid pointer to an array of `semaphoreCount` valid `VkSemaphore` handles
- `pValues` **must** be a valid pointer to an array of `semaphoreCount uint64_t` values
- `semaphoreCount` **must** be greater than `0`

Bits which **can** be set in `VkSemaphoreWaitInfoKHR::flags`, specifying additional parameters of a semaphore wait operation, are:

```
typedef enum VkSemaphoreWaitFlagBitsKHR {
    VK_SEMAPHORE_WAIT_ANY_BIT_KHR = 0x00000001,
    VK_SEMAPHORE_WAIT_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkSemaphoreWaitFlagBitsKHR;
```

- `VK_SEMAPHORE_WAIT_ANY_BIT_KHR` specifies that the semaphore wait condition is that at least one of the semaphores in `VkSemaphoreWaitInfoKHR::pSemaphores` has reached the value specified by the corresponding element of `VkSemaphoreWaitInfoKHR::pValues`. If `VK_SEMAPHORE_WAIT_ANY_BIT_KHR` is not set, the semaphore wait condition is that all of the semaphores in `VkSemaphoreWaitInfoKHR ::pSemaphores` have reached the value specified by the corresponding element of `VkSemaphoreWaitInfoKHR::pValues`.

```
typedef VkFlags VkSemaphoreWaitFlagsKHR;
```

`VkSemaphoreWaitFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSemaphoreWaitFlagBitsKHR`.

To signal a semaphore created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` with a particular counter value, on the host, call:

```
VkResult vkSignalSemaphoreKHR(
    VkDevice                                     device,
    const VkSemaphoreSignalInfoKHR* pSignalInfo);
```

- `device` is the logical device that owns the semaphore.
- `pSignalInfo` is a pointer to an instance of the `VkSemaphoreSignalInfoKHR` structure containing information about the signal operation.

When `vkSignalSemaphoreKHR` is executed on the host, it defines and immediately executes a

[semaphore signal operation](#) which sets the timeline semaphore to the given value.

The first synchronization scope is defined by the host execution model, but includes execution of `vkSignalSemaphoreKHR` on the host and anything that happened-before it.

The second synchronization scope is empty.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pSignalInfo` **must** be a valid pointer to a valid `VkSemaphoreCreateInfoKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkSemaphoreCreateInfoKHR` structure is defined as:

```
typedef struct VkSemaphoreCreateInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSemaphore        semaphore;
    uint64_t           value;
} VkSemaphoreCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `semaphore` is the handle of the semaphore to signal.
- `value` is the value to signal.

## Valid Usage

- `semaphore` **must** have been created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR`
- `value` **must** have a value greater than the current value of the semaphore
- `value` **must** be less than the value of any pending semaphore signal operations
- `value` **must** have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on `semaphore` by more than `maxTimelineSemaphoreValueDifference`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO_KHR`
- `pNext` **must** be `NULL`
- `semaphore` **must** be a valid `VkSemaphore` handle

### 6.4.5. Importing Semaphore Payloads

Applications **can** import a semaphore payload into an existing semaphore using an external semaphore handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the implementation **must** restore the semaphore to its prior permanent state after submitting the next semaphore wait operation. Performing a subsequent temporary import on a semaphore before performing a semaphore wait has no effect on this requirement; the next wait submitted on the semaphore **must** still restore its last permanent state. A permanent payload import behaves as if the target semaphore was destroyed, and a new semaphore was created with the same handle but the imported payload. Because importing a semaphore payload temporarily or permanently detaches the existing payload from a semaphore, similar usage restrictions to those applied to `vkDestroySemaphore` are applied to any command that imports a semaphore payload. Which of these import types is used is referred to as the import operation's *permanence*. Each handle type supports either one or both types of permanence.

The implementation **must** perform the import operation by either referencing or copying the payload referred to by the specified external semaphore handle, depending on the handle's type. The import method used is referred to as the handle type's *transference*. When using handle types with reference transference, importing a payload to a semaphore adds the semaphore to the set of all semaphores sharing that payload. This set includes the semaphore from which the payload was exported. Semaphore signaling and waiting operations performed on any semaphore in the set **must** behave as if the set were a single semaphore. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Semaphore signaling and waiting operations performed on the target of copy imports **must** not affect any other semaphore or payload.

Export operations have the same transference as the specified handle type's import operations.

Additionally, exporting a semaphore payload to a handle with copy transference has the same side effects on the source semaphore's payload as executing a semaphore wait operation. If the semaphore was using a temporarily imported payload, the semaphore's prior permanent payload will be restored.

*Note*



The tables [Handle Types Supported by VkImportSemaphoreWin32HandleInfoKHR](#) and [Handle Types Supported by VkImportSemaphoreFdInfoKHR](#) define the permanence and transference of each handle type.

[External synchronization](#) allows implementations to modify an object's internal state, i.e. payload, without internal synchronization. However, for semaphores sharing a payload across processes, satisfying the external synchronization requirements of [VkSemaphore](#) parameters as if all semaphores in the set were the same object is sometimes infeasible. Satisfying the [wait operation state requirements](#) would similarly require impractical coordination or levels of trust between processes. Therefore, these constraints only apply to a specific semaphore handle, not to its payload. For distinct semaphore objects which share a payload, if the semaphores are passed to separate queue submission commands concurrently, behavior will be as if the commands were called in an arbitrary sequential order. If the [wait operation state requirements](#) are violated for the shared payload by a queue submission command, or if a signal operation is queued for a shared payload that is already signaled or has a pending signal operation, effects **must** be limited to one or more of the following:

- Returning [VK\\_ERROR\\_INITIALIZATION\\_FAILED](#) from the command which resulted in the violation.
- Losing the logical device on which the violation occurred immediately or at a future time, resulting in a [VK\\_ERROR\\_DEVICE\\_LOST](#) error from subsequent commands, including the one causing the violation.
- Continuing execution of the violating command or operation as if the semaphore wait completed successfully after an implementation-dependent timeout. In this case, the state of the payload becomes undefined, and future operations on semaphores sharing the payload will be subject to these same rules. The semaphore **must** be destroyed or have its payload replaced by an import operation to again have a well-defined state.

*Note*



These rules allow processes to synchronize access to shared memory without trusting each other. However, such processes must still be cautious not to use the shared semaphore for more than synchronizing access to the shared memory. For example, a process should not use a shared semaphore as part of an execution dependency chain that, when complete, leads to objects being destroyed, if it does not trust other processes sharing the semaphore payload.

When a semaphore is using an imported payload, its [VkExportSemaphoreCreateInfo::handleTypes](#) value is that specified when creating the semaphore from which the payload was exported, rather than that specified when creating the semaphore. Additionally, [VkExternalSemaphoreProperties::exportFromImportedHandleTypes](#) restricts which handle types **can** be exported from such a semaphore based on the specific handle type used to import the current payload. Passing a semaphore to [vkAcquireNextImageKHR](#) is equivalent to temporarily importing a semaphore

payload to that semaphore.

*Note*

Because the exportable handle types of an imported semaphore correspond to its current imported payload, and [vkAcquireNextImageKHR](#) behaves the same as a temporary import operation for which the source semaphore is opaque to the application, applications have no way of determining whether any external handle types **can** be exported from a semaphore in this state. Therefore, applications **must** not attempt to export external handles from semaphores using a temporarily imported payload from [vkAcquireNextImageKHR](#).



When importing a semaphore payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations **must** perform sufficient validation of external handles to ensure that the operation results in a valid semaphore which will not cause program termination, device loss, queue stalls, or corruption of other resources when used as allowed according to its import parameters, and excepting those side effects allowed for violations of the [valid semaphore state for wait operations](#) rules. If the external handle provided does not meet these requirements, the implementation **must** fail the semaphore payload import operation with the error code [VK\\_ERROR\\_INVALID\\_EXTERNAL\\_HANDLE](#).

In addition, when importing a semaphore payload that is not compatible with the payload type corresponding to the [VkSemaphoreTypeKHR](#) the semaphore was created with, the implementation **may** fail the semaphore payload import operation with the error code [VK\\_ERROR\\_INVALID\\_EXTERNAL\\_HANDLE](#).

*Note*

As the introduction of the external semaphore handle type [VK\\_EXTERNAL\\_SEMAPHORE\\_HANDLE\\_TYPE\\_D3D12\\_FENCE\\_BIT](#) predates that of timeline semaphores, support for importing semaphore payloads from external handles of that type into semaphores created (implicitly or explicitly) with a [VkSemaphoreTypeKHR](#) of [VK\\_SEMAPHORE\\_TYPE\\_BINARY\\_KHR](#) is preserved for backwards compatibility. However, applications **should** prefer importing such handle types into semaphores created with a [VkSemaphoreTypeKHR](#) of [VK\\_SEMAPHORE\\_TYPE\\_TIMELINE\\_KHR](#).



To import a semaphore payload from a Windows handle, call:

```
VkResult vkImportSemaphoreWin32HandleKHR(  
    VkDevice                               device,  
    const VkImportSemaphoreWin32HandleInfoKHR* pImportSemaphoreWin32HandleInfo);
```

- `device` is the logical device that created the semaphore.
- `pImportSemaphoreWin32HandleInfo` is a pointer to a [VkImportSemaphoreWin32HandleInfoKHR](#) structure specifying the semaphore and import parameters.

Importing a semaphore payload from Windows handles does not transfer ownership of the handle to the Vulkan implementation. For handle types defined as NT handles, the application **must**

release ownership using the `CloseHandle` system call when the handle is no longer needed.

Applications **can** import the same semaphore payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pImportSemaphoreWin32HandleInfo` **must** be a valid pointer to a valid `VkImportSemaphoreWin32HandleInfoKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkImportSemaphoreWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkImportSemaphoreWin32HandleInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkSemaphore                semaphore;
    VkSemaphoreImportFlags      flags;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
    HANDLE                     handle;
    LPCWSTR                   name;
} VkImportSemaphoreWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `semaphore` is the semaphore into which the payload will be imported.
- `flags` is a bitmask of `VkSemaphoreImportFlagBits` specifying additional parameters for the semaphore payload import operation.
- `handleType` specifies the type of `handle`.
- `handle` is the external handle to import, or `NULL`.
- `name` is a null-terminated UTF-16 string naming the underlying synchronization primitive to import, or `NULL`.

The handle types supported by `handleType` are:

Table 8. Handle Types Supported by [VkImportSemaphoreWin32HandleInfoKHR](#)

Handle Type	Transference	Permanence Supported
<code>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT</code>	Reference	Temporary,Permanent
<code>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</code>	Reference	Temporary,Permanent
<code>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT</code>	Reference	Temporary,Permanent

## Valid Usage

- `handleType` **must** be a value included in the Handle Types Supported by [VkImportSemaphoreWin32HandleInfoKHR](#) table.
- If `handleType` is not `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT` or `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT`, `name` **must** be `NULL`.
- If `handleType` is not `0` and `name` is `NULL`, `name` **must** name a valid synchronization primitive of the type specified by `handleType`.
- If `handleType` is not `0` and `name` is `NULL`, `handle` **must** be a valid handle of the type specified by `handleType`.
- If `handle` is not `NULL`, `name` **must** be `NULL`.
- If `handle` is not `NULL`, it **must** obey any requirements listed for `handleType` in [external semaphore handle types compatibility](#).
- If `name` is not `NULL`, it **must** obey any requirements listed for `handleType` in [external semaphore handle types compatibility](#).
- If `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT` or `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT`, the `VkSemaphoreCreateInfo`::`flags` field **must** match that of the semaphore from which `handle` or `name` was exported.
- If `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT` or `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT`, the `VkSemaphoreTypeCreateInfoKHR`::`semaphoreType` field **must** match that of the semaphore from which `handle` or `name` was exported.
- If `flags` contains `VK_SEMAPHORE_IMPORT_TEMPORARY_BIT`, the `VkSemaphoreTypeCreateInfoKHR`::`semaphoreType` field of the semaphore from which `handle` or `name` was exported **must** not be `VK_SEMAPHORE_TYPE_TIMELINE_KHR`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR`
- `pNext` **must** be `NULL`
- `semaphore` **must** be a valid `VkSemaphore` handle
- `flags` **must** be a valid combination of `VkSemaphoreImportFlagBits` values
- If `handleType` is not `0`, `handleType` **must** be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

## Host Synchronization

- Host access to `semaphore` **must** be externally synchronized

To import a semaphore payload from a POSIX file descriptor, call:

```
VkResult vkImportSemaphoreFdKHR(  
    VkDevice device,  
    const VkImportSemaphoreFdInfoKHR* pImportSemaphoreFdInfo);
```

- `device` is the logical device that created the semaphore.
- `pImportSemaphoreFdInfo` is a pointer to a `VkImportSemaphoreFdInfoKHR` structure specifying the semaphore and import parameters.

Importing a semaphore payload from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application **must** not perform any operations on the file descriptor after a successful import.

Applications **can** import the same semaphore payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

## Valid Usage

- `semaphore` **must** not be associated with any queue command that has not yet completed execution on that queue

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pImportSemaphoreFdInfo` **must** be a valid pointer to a valid `VkImportSemaphoreFdInfoKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkImportSemaphoreFdInfoKHR` structure is defined as:

```
typedef struct VkImportSemaphoreFdInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkSemaphore               semaphore;
    VkSemaphoreImportFlags    flags;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
    int                       fd;
} VkImportSemaphoreFdInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `semaphore` is the semaphore into which the payload will be imported.
- `flags` is a bitmask of `VkSemaphoreImportFlagBits` specifying additional parameters for the semaphore payload import operation.
- `handleType` specifies the type of `fd`.
- `fd` is the external handle to import.

The handle types supported by `handleType` are:

Table 9. Handle Types Supported by `VkImportSemaphoreFdInfoKHR`

Handle Type	Transference	Permanence Supported
<code>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT</code>	Reference	Temporary,Permanent
<code>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT</code>	Copy	Temporary

## Valid Usage

- `handleType` **must** be a value included in the Handle Types Supported by [VkImportSemaphoreFdInfoKHR](#) table.
- `fd` **must** obey any requirements listed for `handleType` in [external semaphore handle types compatibility](#).
- If `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`, the [VkSemaphoreCreateInfo::flags](#) field **must** match that of the semaphore from which `fd` was exported.
- If `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`, the [VkSemaphoreTypeCreateInfoKHR::semaphoreType](#) field **must** match that of the semaphore from which `fd` was exported.
- If `flags` contains `VK_SEMAPHORE_IMPORT_TEMPORARY_BIT`, the [VkSemaphoreTypeCreateInfoKHR::semaphoreType](#) field of the semaphore from which `fd` was exported **must** not be `VK_SEMAPHORE_TYPE_TIMELINE_KHR`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHR`
- `pNext` **must** be `NULL`
- `semaphore` **must** be a valid `VkSemaphore` handle
- `flags` **must** be a valid combination of `VkSemaphoreImportFlagBits` values
- `handleType` **must** be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

## Host Synchronization

- Host access to `semaphore` **must** be externally synchronized

Additional parameters of a semaphore import operation are specified by [VkImportSemaphoreWin32HandleInfoKHR::flags](#) or [VkImportSemaphoreFdInfoKHR::flags](#). Bits which can be set include:

```
typedef enum VkSemaphoreImportFlagBits {
    VK_SEMAPHORE_IMPORT_TEMPORARY_BIT = 0x00000001,
    VK_SEMAPHORE_IMPORT_TEMPORARY_BIT_KHR = VK_SEMAPHORE_IMPORT_TEMPORARY_BIT,
    VK_SEMAPHORE_IMPORT_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkSemaphoreImportFlagBits;
```

or the equivalent

```
typedef VkSemaphoreImportFlagBits VkSemaphoreImportFlagBitsKHR;
```

These bits have the following meanings:

- **VK\_SEMAPHORE\_IMPORT\_TEMPORARY\_BIT** specifies that the semaphore payload will be imported only temporarily, as described in [Importing Semaphore Payloads](#), regardless of the permanence of **handleType**.

```
typedef VkFlags VkSemaphoreImportFlags;
```

or the equivalent

```
typedef VkSemaphoreImportFlags VkSemaphoreImportFlagsKHR;
```

**VkSemaphoreImportFlags** is a bitmask type for setting a mask of zero or more [VkSemaphoreImportFlagBits](#).

## 6.5. Events

Events are a synchronization primitive that **can** be used to insert a fine-grained dependency between commands submitted to the same queue, or between the host and a queue. Events **must** not be used to insert a dependency between commands submitted to different queues. Events have two states - signaled and unsignaled. An application **can** signal an event, or unsignal it, on either the host or the device. A device **can** wait for an event to become signaled before executing further operations. No command exists to wait for an event to become signaled on the host, but the current state of an event **can** be queried.

Events are represented by **VkEvent** handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkEvent)
```

To create an event, call:

```
VkResult vkCreateEvent(  
    VkDevice                                     device,  
    const VkEventCreateInfo*                    pCreateInfo,  
    const VkAllocationCallbacks*                pAllocator,  
    VkEvent*                                    pEvent);
```

- **device** is the logical device that creates the event.
- **pCreateInfo** is a pointer to a **VkEventCreateInfo** structure containing information about how the event is to be created.
- **pAllocator** controls host memory allocation as described in the [Memory Allocation](#) chapter.

- `pEvent` is a pointer to a handle in which the resulting event object is returned.

When created, the event object is in the unsignaled state.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkEventCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pEvent` **must** be a valid pointer to a `VkEvent` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkEventCreateInfo` structure is defined as:

```
typedef struct VkEventCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkEventCreateFlags flags;
} VkEventCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EVENT_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

```
typedef VkFlags VkEventCreateFlags;
```

`VkEventCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To destroy an event, call:

```
void vkDestroyEvent(  
    VkDevice device,  
    VkEvent event,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the event.
- `event` is the handle of the event to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `event` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `event` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `event` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `event` is not `VK_NULL_HANDLE`, `event` **must** be a valid `VkEvent` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `event` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `event` **must** be externally synchronized

To query the state of an event from the host, call:

```
VkResult vkGetEventStatus(  
    VkDevice device,  
    VkEvent event);
```

- `device` is the logical device that owns the event.
- `event` is the handle of the event to query.

Upon success, `vkGetEventStatus` returns the state of the event object with the following return codes:

Table 10. Event Object Status Codes

Status	Meaning
<code>VK_EVENT_SET</code>	The event specified by <code>event</code> is signaled.
<code>VK_EVENT_RESET</code>	The event specified by <code>event</code> is unsignaled.

If a `vkCmdSetEvent` or `vkCmdResetEvent` command is in a command buffer that is in the `pending` state, then the value returned by this command **may** immediately be out of date.

The state of an event **can** be updated by the host. The state of the event is immediately changed, and subsequent calls to `vkGetEventStatus` will return the new state. If an event is already in the requested state, then updating it to the same state has no effect.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `event` **must** be a valid `VkEvent` handle
- `event` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_EVENT_SET`
- `VK_EVENT_RESET`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

To set the state of an event to signaled from the host, call:

```
VkResult vkSetEvent(  
    VkDevice           device,  
    VkEvent            event);
```

- `device` is the logical device that owns the event.
- `event` is the event to set.

When `vkSetEvent` is executed on the host, it defines an *event signal operation* which sets the event to the signaled state.

If `event` is already in the signaled state when `vkSetEvent` is executed, then `vkSetEvent` has no effect, and no event signal operation occurs.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `event` **must** be a valid `VkEvent` handle
- `event` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `event` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

To set the state of an event to unsignaled from the host, call:

```
VkResult vkResetEvent(  
    VkDevice                device,  
    VkEvent                 event);
```

- `device` is the logical device that owns the event.
- `event` is the event to reset.

When `vkResetEvent` is executed on the host, it defines an *event unsignal operation* which resets the event to the unsignaled state.

If `event` is already in the unsignaled state when `vkResetEvent` is executed, then `vkResetEvent` has no effect, and no event unsignal operation occurs.

## Valid Usage

- `event` **must** not be waited on by a `vkCmdWaitEvents` command that is currently executing

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `event` **must** be a valid `VkEvent` handle
- `event` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `event` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The state of an event **can** also be updated on the device by commands inserted in command buffers.

To set the state of an event to signaled from a device, call:

```
void vkCmdSetEvent(  
    VkCommandBuffer  
    VkEvent  
    VkPipelineStageFlags  
        commandBuffer,  
        event,  
        stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be signaled.
- `stageMask` specifies the `source stage mask` used to determine when the `event` is signaled.

When `vkCmdSetEvent` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event signal operation which sets the event to the signaled state.

The first `synchronization scope` includes all commands that occur earlier in `submission order`. The synchronization scope is limited to operations on the pipeline stages determined by the `source stage mask` specified by `stageMask`.

The second `synchronization scope` includes only the event signal operation.

If `event` is already in the signaled state when `vkCmdSetEvent` is executed on the device, then

`vkCmdSetEvent` has no effect, no event signal operation occurs, and no execution dependency is generated.

## Valid Usage

- `stageMask` **must** not include `VK_PIPELINE_STAGE_HOST_BIT`
- If the `geometry shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- `commandBuffer`'s current device mask **must** include exactly one physical device.
- If the `mesh shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `event` **must** be a valid `VkEvent` handle
- `stageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `stageMask` **must** not be `0`
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- Both of `commandBuffer`, and `event` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics Compute	

To set the state of an event to unsignaled from a device, call:

```
void vkCmdResetEvent(  
    VkCommandBuffer  
    VkEvent  
    VkPipelineStageFlags  
        commandBuffer,  
        event,  
        stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be unsignaled.
- `stageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `source stage mask` used to determine when the `event` is unsignaled.

When `vkCmdResetEvent` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event unsignal operation which resets the event to the unsignaled state.

The first `synchronization scope` includes all commands that occur earlier in `submission order`. The synchronization scope is limited to operations on the pipeline stages determined by the `source stage mask` specified by `stageMask`.

The second `synchronization scope` includes only the event unsignal operation.

If `event` is already in the unsignaled state when `vkCmdResetEvent` is executed on the device, then `vkCmdResetEvent` has no effect, no event unsignal operation occurs, and no execution dependency is generated.

## Valid Usage

- `stageMask` **must** not include `VK_PIPELINE_STAGE_HOST_BIT`
- If the `geometry shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- When this command executes, `event` **must** not be waited on by a `vkCmdWaitEvents` command that is currently executing
- `commandBuffer`'s current device mask **must** include exactly one physical device.
- If the `mesh shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `event` **must** be a valid `VkEvent` handle
- `stageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `stageMask` **must** not be `0`
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- Both of `commandBuffer`, and `event` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics Compute	

To wait for one or more events to enter the signaled state on a device, call:

```
void vkCmdWaitEvents(  
    VkCommandBuffer  
    uint32_t  
    const VkEvent*  
    VkPipelineStageFlags  
    VkPipelineStageFlags  
    uint32_t  
    const VkMemoryBarrier*  
    uint32_t  
    const VkBufferMemoryBarrier*  
    uint32_t  
    const VkImageMemoryBarrier*  
        commandBuffer,  
        eventCount,  
        pEvents,  
        srcStageMask,  
        dstStageMask,  
        memoryBarrierCount,  
        pMemoryBarriers,  
        bufferMemoryBarrierCount,  
        pBufferMemoryBarriers,  
        imageMemoryBarrierCount,  
        pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `eventCount` is the length of the `pEvents` array.
- `pEvents` is a pointer to an array of event object handles to wait on.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `source stage mask`.
- `dstStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `destination stage mask`.
- `memoryBarrierCount` is the length of the `pMemoryBarriers` array.
- `pMemoryBarriers` is a pointer to an array of `VkMemoryBarrier` structures.
- `bufferMemoryBarrierCount` is the length of the `pBufferMemoryBarriers` array.
- `pBufferMemoryBarriers` is a pointer to an array of `VkBufferMemoryBarrier` structures.
- `imageMemoryBarrierCount` is the length of the `pImageMemoryBarriers` array.
- `pImageMemoryBarriers` is a pointer to an array of `VkImageMemoryBarrier` structures.

When `vkCmdWaitEvents` is submitted to a queue, it defines a memory dependency between prior event signal operations on the same queue or the host, and subsequent commands. `vkCmdWaitEvents` **must** not be used to wait on event signal operations occurring on other queues.

The first synchronization scope only includes event signal operations that operate on members of `pEvents`, and the operations that happened-before the event signal operations. Event signal operations performed by `vkCmdSetEvent` that occur earlier in `submission order` are included in the first synchronization scope, if the `logically latest` pipeline stage in their `stageMask` parameter is `logically earlier` than or equal to the `logically latest` pipeline stage in `srcStageMask`. Event signal

operations performed by `vkSetEvent` are only included in the first synchronization scope if `VK_PIPELINE_STAGE_HOST_BIT` is included in `srcStageMask`.

The second synchronization scope includes all commands that occur later in submission order. The second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by `dstStageMask`.

The first access scope is limited to access in the pipeline stages determined by the source stage mask specified by `srcStageMask`. Within that, the first access scope only includes the first access scopes defined by elements of the `pMemoryBarriers`, `pBufferMemoryBarriers` and `pImageMemoryBarriers` arrays, which each define a set of memory barriers. If no memory barriers are specified, then the first access scope includes no accesses.

The second access scope is limited to access in the pipeline stages determined by the destination stage mask specified by `dstStageMask`. Within that, the second access scope only includes the second access scopes defined by elements of the `pMemoryBarriers`, `pBufferMemoryBarriers` and `pImageMemoryBarriers` arrays, which each define a set of memory barriers. If no memory barriers are specified, then the second access scope includes no accesses.

*Note*



`vkCmdWaitEvents` is used with `vkCmdSetEvent` to define a memory dependency between two sets of action commands, roughly in the same way as pipeline barriers, but split into two commands such that work between the two may execute unhindered.

Unlike `vkCmdPipelineBarrier`, a queue family ownership transfer cannot be performed using `vkCmdWaitEvents`.

*Note*



Applications should be careful to avoid race conditions when using events. There is no direct ordering guarantee between a `vkCmdResetEvent` command and a `vkCmdWaitEvents` command submitted after it, so some other execution dependency must be included between these commands (e.g. a semaphore).

## Valid Usage

- `srcStageMask` **must** be the bitwise OR of the `stageMask` parameter used in previous calls to `vkCmdSetEvent` with any of the members of `pEvents` and `VK_PIPELINE_STAGE_HOST_BIT` if any of the members of `pEvents` was set using `vkSetEvent`
- If the `geometry shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `geometry shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If `pEvents` includes one or more events that will be signaled by `vkSetEvent` after `commandBuffer` has been submitted to a queue, then `vkCmdWaitEvents` **must** not be called inside a render pass instance
- Any pipeline stage included in `srcStageMask` or `dstStageMask` **must** be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the [table of supported pipeline stages](#).
- Each element of `pMemoryBarriers`, `pBufferMemoryBarriers` or `pImageMemoryBarriers` **must** not have any access flag included in its `srcAccessMask` member if that bit is not supported by any of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#).
- Each element of `pMemoryBarriers`, `pBufferMemoryBarriers` or `pImageMemoryBarriers` **must** not have any access flag included in its `dstAccessMask` member if that bit is not supported by any of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#).
- The `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members of any element of `pBufferMemoryBarriers` or `pImageMemoryBarriers` **must** be equal.
- `commandBuffer`'s current device mask **must** include exactly one physical device.
- If the `mesh shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
- If the `mesh shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
- The `srcAccessMask` member of each element of `pMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)

- The `dstAccessMask` member of each element of `pMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)
- The `srcAccessMask` member of each element of `pBufferMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)
- The `dstAccessMask` member of each element of `pBufferMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)
- The `srcAccessMask` member of each element of `pImageMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)
- The `dstAccessMask` member of any element of `pImageMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pEvents` **must** be a valid pointer to an array of `eventCount` valid `VkEvent` handles
- `srcStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `srcStageMask` **must** not be `0`
- `dstStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `dstStageMask` **must** not be `0`
- If `memoryBarrierCount` is not `0`, `pMemoryBarriers` **must** be a valid pointer to an array of `memoryBarrierCount` valid `VkMemoryBarrier` structures
- If `bufferMemoryBarrierCount` is not `0`, `pBufferMemoryBarriers` **must** be a valid pointer to an array of `bufferMemoryBarrierCount` valid `VkBufferMemoryBarrier` structures
- If `imageMemoryBarrierCount` is not `0`, `pImageMemoryBarriers` **must** be a valid pointer to an array of `imageMemoryBarrierCount` valid `VkImageMemoryBarrier` structures
- `commandBuffer` **must** be in the [recording state](#)
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- `eventCount` **must** be greater than `0`
- Both of `commandBuffer`, and the elements of `pEvents` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

## 6.6. Pipeline Barriers

`vkCmdPipelineBarrier` is a synchronization command that inserts a dependency between commands submitted to the same queue, or between commands in the same subpass.

To record a pipeline barrier, call:

```
void vkCmdPipelineBarrier(  
    VkCommandBuffer  
    VkPipelineStageFlags  
    VkPipelineStageFlags  
    VkDependencyFlags  
    uint32_t  
    const VkMemoryBarrier*  
    uint32_t  
    const VkBufferMemoryBarrier*  
    uint32_t  
    const VkImageMemoryBarrier*  
                                commandBuffer,  
                                srcStageMask,  
                                dstStageMask,  
                                dependencyFlags,  
                                memoryBarrierCount,  
                                pMemoryBarriers,  
                                bufferMemoryBarrierCount,  
                                pBufferMemoryBarriers,  
                                imageMemoryBarrierCount,  
                                pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `source stage mask`.
- `dstStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `destination stage mask`.
- `dependencyFlags` is a bitmask of `VkDependencyFlagBits` specifying how execution and memory dependencies are formed.
- `memoryBarrierCount` is the length of the `pMemoryBarriers` array.
- `pMemoryBarriers` is a pointer to an array of `VkMemoryBarrier` structures.
- `bufferMemoryBarrierCount` is the length of the `pBufferMemoryBarriers` array.
- `pBufferMemoryBarriers` is a pointer to an array of `VkBufferMemoryBarrier` structures.

- `imageMemoryBarrierCount` is the length of the `pImageMemoryBarriers` array.
- `pImageMemoryBarriers` is a pointer to an array of `VkImageMemoryBarrier` structures.

When `vkCmdPipelineBarrier` is submitted to a queue, it defines a memory dependency between commands that were submitted before it, and those submitted after it.

If `vkCmdPipelineBarrier` was recorded outside a render pass instance, the first synchronization scope includes all commands that occur earlier in `submission order`. If `vkCmdPipelineBarrier` was recorded inside a render pass instance, the first synchronization scope includes only commands that occur earlier in `submission order` within the same subpass. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the `source stage mask` specified by `srcStageMask`.

If `vkCmdPipelineBarrier` was recorded outside a render pass instance, the second synchronization scope includes all commands that occur later in `submission order`. If `vkCmdPipelineBarrier` was recorded inside a render pass instance, the second synchronization scope includes only commands that occur later in `submission order` within the same subpass. In either case, the second synchronization scope is limited to operations on the pipeline stages determined by the `destination stage mask` specified by `dstStageMask`.

The first `access scope` is limited to access in the pipeline stages determined by the `source stage mask` specified by `srcStageMask`. Within that, the first access scope only includes the first access scopes defined by elements of the `pMemoryBarriers`, `pBufferMemoryBarriers` and `pImageMemoryBarriers` arrays, which each define a set of `memory barriers`. If no memory barriers are specified, then the first access scope includes no accesses.

The second `access scope` is limited to access in the pipeline stages determined by the `destination stage mask` specified by `dstStageMask`. Within that, the second access scope only includes the second access scopes defined by elements of the `pMemoryBarriers`, `pBufferMemoryBarriers` and `pImageMemoryBarriers` arrays, which each define a set of `memory barriers`. If no memory barriers are specified, then the second access scope includes no accesses.

If `dependencyFlags` includes `VK_DEPENDENCY_BY_REGION_BIT`, then any dependency between framebuffer-space pipeline stages is `framebuffer-local` - otherwise it is `framebuffer-global`.

## Valid Usage

- If the `geometry shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `geometry shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If `vkCmdPipelineBarrier` is called within a render pass instance, the render pass **must** have been created with at least one `VkSubpassDependency` instance in `VkRenderPassCreateInfo::pDependencies` that expresses a dependency from the current subpass to itself, and for which `srcStageMask` contains a subset of the bit values in `VkSubpassDependency::srcStageMask`, `dstStageMask` contains a subset of the bit values in `VkSubpassDependency::dstStageMask`, `dependencyFlags` is equal to `VkSubpassDependency::dependencyFlags`, `srcAccessMask` member of each element of `pMemoryBarriers` and `pImageMemoryBarriers` contains a subset of the bit values in `VkSubpassDependency::srcAccessMask`, and `dstAccessMask` member of each element of `pMemoryBarriers` and `pImageMemoryBarriers` contains a subset of the bit values in `VkSubpassDependency::dstAccessMask`
- If `vkCmdPipelineBarrier` is called within a render pass instance, `bufferMemoryBarrierCount` **must** be `0`
- If `vkCmdPipelineBarrier` is called within a render pass instance, the `image` member of any element of `pImageMemoryBarriers` **must** be equal to one of the elements of `pAttachments` that the current `framebuffer` was created with, that is also referred to by one of the elements of the `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment` members of the `VkSubpassDescription` instance or by the `pDepthStencilResolveAttachment` member of the `VkSubpassDescriptionDepthStencilResolveKHR` structure that the current subpass was created with
- If `vkCmdPipelineBarrier` is called within a render pass instance, the `oldLayout` and `newLayout` members of any element of `pImageMemoryBarriers` **must** be equal to the `layout` member of an element of the `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment` members of the `VkSubpassDescription` instance or by the `pDepthStencilResolveAttachment` member of the `VkSubpassDescriptionDepthStencilResolveKHR` structure that the current subpass was created with, that refers to the same `image`
- If `vkCmdPipelineBarrier` is called within a render pass instance, the `oldLayout` and `newLayout` members of an element of `pImageMemoryBarriers` **must** be equal
- If `vkCmdPipelineBarrier` is called within a render pass instance, the `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members of any element of `pImageMemoryBarriers` **must** be `VK_QUEUE_FAMILY_IGNORED`
- Any pipeline stage included in `srcStageMask` or `dstStageMask` **must** be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the

`VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the [table of supported pipeline stages](#)

- If `vkCmdPipelineBarrier` is called outside of a render pass instance, `dependencyFlags` **must** not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`
- If the `mesh shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
- If the `mesh shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
- The `srcAccessMask` member of each element of `pMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)
- The `dstAccessMask` member of each element of `pMemoryBarriers` **must** only include access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)
- For any element of `pBufferMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `srcQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `srcAccessMask` member **must** only contain access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)
- For any element of `pBufferMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `dstQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `dstAccessMask` member **must** only contain access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)
- For any element of `pImageMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `srcQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `srcAccessMask` member **must** only contain access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)
- For any element of `pImageMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `dstQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `dstAccessMask` member **must** only contain access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `srcStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `srcStageMask` **must** not be `0`
- `dstStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `dstStageMask` **must** not be `0`
- `dependencyFlags` **must** be a valid combination of `VkDependencyFlagBits` values
- If `memoryBarrierCount` is not `0`, `pMemoryBarriers` **must** be a valid pointer to an array of `memoryBarrierCount` valid `VkMemoryBarrier` structures
- If `bufferMemoryBarrierCount` is not `0`, `pBufferMemoryBarriers` **must** be a valid pointer to an array of `bufferMemoryBarrierCount` valid `VkBufferMemoryBarrier` structures
- If `imageMemoryBarrierCount` is not `0`, `pImageMemoryBarriers` **must** be a valid pointer to an array of `imageMemoryBarrierCount` valid `VkImageMemoryBarrier` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Transfer Graphics Compute	

Bits which **can** be set in `vkCmdPipelineBarrier::dependencyFlags`, specifying how execution and memory dependencies are formed, are:

```

typedef enum VkDependencyFlagBits {
    VK_DEPENDENCY_BY_REGION_BIT = 0x00000001,
    VK_DEPENDENCY_DEVICE_GROUP_BIT = 0x00000004,
    VK_DEPENDENCY_VIEW_LOCAL_BIT = 0x00000002,
    VK_DEPENDENCY_VIEW_LOCAL_BIT_KHR = VK_DEPENDENCY_VIEW_LOCAL_BIT,
    VK_DEPENDENCY_DEVICE_GROUP_BIT_KHR = VK_DEPENDENCY_DEVICE_GROUP_BIT,
    VK_DEPENDENCY_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkDependencyFlagBits;

```

- **VK\_DEPENDENCY\_BY\_REGION\_BIT** specifies that dependencies will be [framebuffer-local](#).
- **VK\_DEPENDENCY\_VIEW\_LOCAL\_BIT** specifies that a [subpass has more than one view](#).
- **VK\_DEPENDENCY\_DEVICE\_GROUP\_BIT** specifies that dependencies are [non-device-local dependency](#).

```
typedef VkFlags VkDependencyFlags;
```

[VkDependencyFlags](#) is a bitmask type for setting a mask of zero or more [VkDependencyFlagBits](#).

### 6.6.1. Subpass Self-dependency

If [vkCmdPipelineBarrier](#) is called inside a render pass instance, the following restrictions apply. For a given subpass to allow a pipeline barrier, the render pass **must** declare a *self-dependency* from that subpass to itself. That is, there **must** exist a [VkSubpassDependency](#) in the subpass dependency list for the render pass with [srcSubpass](#) and [dstSubpass](#) equal to that subpass index. More than one self-dependency **can** be declared for each subpass.

Self-dependencies **must** only include pipeline stage bits that are graphics stages. If any of the stages in [srcStages](#) are [framebuffer-space stages](#), [dstStages](#) **must** only contain [framebuffer-space stages](#). Additionally, [srcStages](#) **must** not contain [VK\\_PIPELINE\\_STAGE\\_BOTTOM\\_OF\\_PIPE\\_BIT](#) in a self-dependency.

If the source and destination stage masks both include framebuffer-space stages, then [dependencyFlags](#) **must** include [VK\\_DEPENDENCY\\_BY\\_REGION\\_BIT](#). If the subpass has more than one view, then [dependencyFlags](#) **must** include [VK\\_DEPENDENCY\\_VIEW\\_LOCAL\\_BIT](#).

A [vkCmdPipelineBarrier](#) command inside a render pass instance **must** be a *subset* of one of the self-dependencies of the subpass it is used in, meaning that the stage masks and access masks **must** each include only a subset of the bits of the corresponding mask in that self-dependency. If the self-dependency has [VK\\_DEPENDENCY\\_BY\\_REGION\\_BIT](#) or [VK\\_DEPENDENCY\\_VIEW\\_LOCAL\\_BIT](#) set, then so **must** the pipeline barrier. Pipeline barriers within a render pass instance **can** only be types [VkMemoryBarrier](#) or [VkImageMemoryBarrier](#). If a [VkImageMemoryBarrier](#) is used, the image and image subresource range specified in the barrier **must** be a subset of one of the image views used by the framebuffer in the current subpass. Additionally, [oldLayout](#) **must** be equal to [newLayout](#), and both the [srcQueueFamilyIndex](#) and [dstQueueFamilyIndex](#) **must** be [VK\\_QUEUE\\_FAMILY\\_IGNORED](#).

## 6.7. Memory Barriers

*Memory barriers* are used to explicitly control access to buffer and image subresource ranges. Memory barriers are used to [transfer ownership between queue families](#), [change image layouts](#), and define [availability and visibility operations](#). They explicitly define the [access types](#) and buffer and image subresource ranges that are included in the [access scopes](#) of a memory dependency that is created by a synchronization command that includes them.

### 6.7.1. Global Memory Barriers

Global memory barriers apply to memory accesses involving all memory objects that exist at the time of its execution.

The `VkMemoryBarrier` structure is defined as:

```
typedef struct VkMemoryBarrier {  
    VkStructureType    sType;  
    const void*        pNext;  
    VkAccessFlags      srcAccessMask;  
    VkAccessFlags      dstAccessMask;  
} VkMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a [source access mask](#).
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a [destination access mask](#).

The first [access scope](#) is limited to access types in the [source access mask](#) specified by `srcAccessMask`.

The second [access scope](#) is limited to access types in the [destination access mask](#) specified by `dstAccessMask`.

#### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_BARRIER`
- `pNext` **must** be `NULL`
- `srcAccessMask` **must** be a valid combination of `VkAccessFlagBits` values
- `dstAccessMask` **must** be a valid combination of `VkAccessFlagBits` values

### 6.7.2. Buffer Memory Barriers

Buffer memory barriers only apply to memory accesses involving a specific buffer range. That is, a memory dependency formed from a buffer memory barrier is [scoped](#) to access via the specified buffer range. Buffer memory barriers [can](#) also be used to define a [queue family ownership transfer](#)

for the specified buffer range.

The `VkBufferMemoryBarrier` structure is defined as:

```
typedef struct VkBufferMemoryBarrier {  
    VkStructureType    sType;  
    const void*        pNext;  
    VkAccessFlags      srcAccessMask;  
    VkAccessFlags      dstAccessMask;  
    uint32_t           srcQueueFamilyIndex;  
    uint32_t           dstQueueFamilyIndex;  
    VkBuffer           buffer;  
    VkDeviceSize       offset;  
    VkDeviceSize       size;  
} VkBufferMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a [source access mask](#).
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a [destination access mask](#).
- `srcQueueFamilyIndex` is the source queue family for a [queue family ownership transfer](#).
- `dstQueueFamilyIndex` is the destination queue family for a [queue family ownership transfer](#).
- `buffer` is a handle to the buffer whose backing memory is affected by the barrier.
- `offset` is an offset in bytes into the backing memory for `buffer`; this is relative to the base offset as bound to the buffer (see [vkBindBufferMemory](#)).
- `size` is a size in bytes of the affected area of backing memory for `buffer`, or `VK_WHOLE_SIZE` to use the range from `offset` to the end of the buffer.

The first [access scope](#) is limited to access to memory through the specified buffer range, via access types in the [source access mask](#) specified by `srcAccessMask`. If `srcAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT`, memory writes performed by that access type are also made visible, as that access type is not performed through a resource.

The second [access scope](#) is limited to access to memory through the specified buffer range, via access types in the [destination access mask](#) specified by `dstAccessMask`. If `dstAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT` or `VK_ACCESS_HOST_READ_BIT`, available memory writes are also made visible to accesses of those types, as those access types are not performed through a resource.

If `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, and `srcQueueFamilyIndex` is equal to the current queue family, then the memory barrier defines a [queue family release operation](#) for the specified buffer range, and the second access scope includes no access, as if `dstAccessMask` was `0`.

If `dstQueueFamilyIndex` is not equal to `srcQueueFamilyIndex`, and `dstQueueFamilyIndex` is equal to the current queue family, then the memory barrier defines a [queue family acquire operation](#) for the specified buffer range, and the first access scope includes no access, as if `srcAccessMask` was `0`.

## Valid Usage

- `offset` **must** be less than the size of `buffer`
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** be greater than `0`
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** be less than or equal to than the size of `buffer` minus `offset`
- If `buffer` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, at least one of `srcQueueFamilyIndex` and `dstQueueFamilyIndex` **must** be `VK_QUEUE_FAMILY_IGNORED`
- If `buffer` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, and one of `srcQueueFamilyIndex` and `dstQueueFamilyIndex` is `VK_QUEUE_FAMILY_IGNORED`, the other **must** be `VK_QUEUE_FAMILY_IGNORED` or a special queue family reserved for external memory ownership transfers, as described in [Queue Family Ownership Transfer](#).
- If `buffer` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE` and `srcQueueFamilyIndex` is `VK_QUEUE_FAMILY_IGNORED`, `dstQueueFamilyIndex` **must** also be `VK_QUEUE_FAMILY_IGNORED`
- If `buffer` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE` and `srcQueueFamilyIndex` is not `VK_QUEUE_FAMILY_IGNORED`, it **must** be a valid queue family or a special queue family reserved for external memory transfers, as described in [Queue Family Ownership Transfer](#).
- If `buffer` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE` and `dstQueueFamilyIndex` is not `VK_QUEUE_FAMILY_IGNORED`, it **must** be a valid queue family or a special queue family reserved for external memory transfers, as described in [Queue Family Ownership Transfer](#).
- If `buffer` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are not `VK_QUEUE_FAMILY_IGNORED`, at least one of them **must** be the same as the family of the queue that will execute this barrier
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER`
- `pNext` **must** be `NULL`
- `buffer` **must** be a valid `VkBuffer` handle

### 6.7.3. Image Memory Barriers

Image memory barriers only apply to memory accesses involving a specific image subresource range. That is, a memory dependency formed from an image memory barrier is [scoped](#) to access via the specified image subresource range. Image memory barriers [can](#) also be used to define [image layout transitions](#) or a [queue family ownership transfer](#) for the specified image subresource

range.

The `VkImageMemoryBarrier` structure is defined as:

```
typedef struct VkImageMemoryBarrier {
    VkStructureType           sType;
    const void*               pNext;
    VkAccessFlags              srcAccessMask;
    VkAccessFlags              dstAccessMask;
    VkImageLayout                oldLayout;
    VkImageLayout                newLayout;
    uint32_t                     srcQueueFamilyIndex;
    uint32_t                     dstQueueFamilyIndex;
    VkImage                      image;
    VkImageSubresourceRange      subresourceRange;
} VkImageMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a `source access mask`.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a `destination access mask`.
- `oldLayout` is the old layout in an `image layout transition`.
- `newLayout` is the new layout in an `image layout transition`.
- `srcQueueFamilyIndex` is the source queue family for a `queue family ownership transfer`.
- `dstQueueFamilyIndex` is the destination queue family for a `queue family ownership transfer`.
- `image` is a handle to the image affected by this barrier.
- `subresourceRange` describes the `image subresource range` within `image` that is affected by this barrier.

The first `access scope` is limited to access to memory through the specified image subresource range, via access types in the `source access mask` specified by `srcAccessMask`. If `srcAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT`, memory writes performed by that access type are also made visible, as that access type is not performed through a resource.

The second `access scope` is limited to access to memory through the specified image subresource range, via access types in the `destination access mask` specified by `dstAccessMask`. If `dstAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT` or `VK_ACCESS_HOST_READ_BIT`, available memory writes are also made visible to accesses of those types, as those access types are not performed through a resource.

If `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, and `srcQueueFamilyIndex` is equal to the current queue family, then the memory barrier defines a `queue family release operation` for the specified image subresource range, and the second access scope includes no access, as if `dstAccessMask` was `0`.

If `dstQueueFamilyIndex` is not equal to `srcQueueFamilyIndex`, and `dstQueueFamilyIndex` is equal to the

current queue family, then the memory barrier defines a [queue family acquire operation](#) for the specified image subresource range, and the first access scope includes no access, as if `srcAccessMask` was `0`.

If `oldLayout` is not equal to `newLayout`, then the memory barrier defines an [image layout transition](#) for the specified image subresource range.

Layout transitions that are performed via image memory barriers execute in their entirety in [submission order](#), relative to other image layout transitions submitted to the same queue, including those performed by [render passes](#). In effect there is an implicit execution dependency from each such layout transition to all layout transitions previously submitted to the same queue.

The image layout of each image subresource of a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` is dependent on the last sample locations used to render to the image subresource as a depth/stencil attachment, thus when the `image` member of a `VkImageMemoryBarrier` is an image created with this flag the application **can** chain a `VkSampleLocationsInfoEXT` structure to the `pNext` chain of `VkImageMemoryBarrier` to specify the sample locations to use during the image layout transition.

If the `VkSampleLocationsInfoEXT` structure in the `pNext` chain of `VkImageMemoryBarrier` does not match the sample location state last used to render to the image subresource range specified by `subresourceRange` or if no `VkSampleLocationsInfoEXT` structure is in the `pNext` chain of `VkImageMemoryBarrier` then the contents of the given image subresource range becomes undefined as if `oldLayout` would equal `VK_IMAGE_LAYOUT_UNDEFINED`.

If `image` has a multi-planar format and the image is *disjoint*, then including `VK_IMAGE_ASPECT_COLOR_BIT` in the `aspectMask` member of `subresourceRange` is equivalent to including `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, and (for three-plane formats only) `VK_IMAGE_ASPECT_PLANE_2_BIT`.

## Valid Usage

- `oldLayout` **must** be `VK_IMAGE_LAYOUT_UNDEFINED` or the current layout of the image subresources affected by the barrier
- `newLayout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`
- If `image` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, at least one of `srcQueueFamilyIndex` and `dstQueueFamilyIndex` **must** be `VK_QUEUE_FAMILY_IGNORED`
- If `image` was created with a sharing mode of `VK_SHARING_MODE_CONCURRENT`, and one of `srcQueueFamilyIndex` and `dstQueueFamilyIndex` is `VK_QUEUE_FAMILY_IGNORED`, the other **must** be `VK_QUEUE_FAMILY_IGNORED` or a special queue family reserved for external memory transfers, as described in [Queue Family Ownership Transfer](#).
- If `image` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE` and `srcQueueFamilyIndex` is `VK_QUEUE_FAMILY_IGNORED`, `dstQueueFamilyIndex` **must** also be `VK_QUEUE_FAMILY_IGNORED`.
- If `image` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE` and `srcQueueFamilyIndex` is not `VK_QUEUE_FAMILY_IGNORED`, it **must** be a valid queue family or a special queue family reserved for external memory transfers, as described in [Queue Family Ownership Transfer](#).
- If `image` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE` and `dstQueueFamilyIndex` is not `VK_QUEUE_FAMILY_IGNORED`, it **must** be a valid queue family or a special queue family reserved for external memory transfers, as described in [Queue Family Ownership Transfer](#).
- If `image` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are not `VK_QUEUE_FAMILY_IGNORED`, at least one of them **must** be the same as the family of the queue that will execute this barrier
- `subresourceRange.baseMipLevel` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- If `subresourceRange.levelCount` is not `VK_REMAINING_MIP_LEVELS`, `subresourceRange.baseMipLevel + subresourceRange.levelCount` **must** be less than or equal to the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- `subresourceRange.baseArrayLayer` **must** be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- If `subresourceRange.layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, `subresourceRange.baseArrayLayer + subresourceRange.layerCount` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- If `image` has a depth/stencil format with both depth and stencil and the `separateDepthStencilLayouts` feature is enabled, then the `aspectMask` member of `subresourceRange` **must** include either or both `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT`
- If `image` has a depth/stencil format with both depth and stencil and the `separateDepthStencilLayouts` feature is not enabled, then the `aspectMask` member of `subresourceRange` **must** include both `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT`

- If `image` has a single-plane color format or is not *disjoint*, then the `aspectMask` member of `subresourceRange` **must** be `VK_IMAGE_ASPECT_COLOR_BIT`
- If `image` has a multi-planar format and the image is *disjoint*, then the `aspectMask` member of `subresourceRange` **must** include either at least one of `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, and `VK_IMAGE_ASPECT_PLANE_2_BIT`; or **must** include `VK_IMAGE_ASPECT_COLOR_BIT`
- If `image` has a multi-planar format with only two planes, then the `aspectMask` member of `subresourceRange` **must** not include `VK_IMAGE_ASPECT_PLANE_2_BIT`
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` set
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` set
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` set
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then `image` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` set
- If `image` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- If either `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV` then `image` **must** have been created with `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV` set

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkSampleLocationsInfoEXT`
- `oldLayout` **must** be a valid `VkImageLayout` value
- `newLayout` **must** be a valid `VkImageLayout` value
- `image` **must** be a valid `VkImage` handle
- `subresourceRange` **must** be a valid `VkImageSubresourceRange` structure

### 6.7.4. Queue Family Ownership Transfer

Resources created with a `VkSharingMode` of `VK_SHARING_MODE_EXCLUSIVE` **must** have their ownership explicitly transferred from one queue family to another in order to access their content in a well-defined manner on a queue in a different queue family. Resources shared with external APIs or instances using external memory **must** also explicitly manage ownership transfers between local and external queues (or equivalent constructs in external APIs) regardless of the `VkSharingMode` specified when creating them. The special queue family index `VK_QUEUE_FAMILY_EXTERNAL` represents any queue external to the resource's current Vulkan instance, as long as the queue uses the same underlying physical device or device group and uses the same driver version as the resource's `VkDevice`, as indicated by `VkPhysicalDeviceIDProperties::deviceUUID` and `VkPhysicalDeviceIDProperties::driverUUID`. The special queue family index `VK_QUEUE_FAMILY_FOREIGN_EXT` represents any queue external to the resource's current Vulkan instance, regardless of the queue's underlying physical device or driver version. This includes, for example, queues for fixed-function image processing devices, media codec devices, and display devices, as well as all queues that use the same underlying physical device (or device group) and driver version as the resource's `VkDevice`. If memory dependencies are correctly expressed between uses of such a resource between two queues in different families, but no ownership transfer is defined, the contents of that resource are undefined for any read accesses performed by the second queue family.

*Note*



If an application does not need the contents of a resource to remain valid when transferring from one queue family to another, then the ownership transfer **should** be skipped.

*Note*



Applications should expect transfers to/from `VK_QUEUE_FAMILY_FOREIGN_EXT` to be more expensive than transfers to/from `VK_QUEUE_FAMILY_EXTERNAL_KHR`.

A queue family ownership transfer consists of two distinct parts:

1. Release exclusive ownership from the source queue family
2. Acquire exclusive ownership for the destination queue family

An application **must** ensure that these operations occur in the correct order by defining an execution dependency between them, e.g. using a semaphore.

A *release operation* is used to release exclusive ownership of a range of a buffer or image subresource range. A release operation is defined by executing a [buffer memory barrier](#) (for a buffer range) or an [image memory barrier](#) (for an image subresource range) using `vkCmdPipelineBarrier`, on a queue from the source queue family. The `srcQueueFamilyIndex` parameter of the barrier **must** be set to the source queue family index, and the `dstQueueFamilyIndex` parameter to the destination queue family index. `dstAccessMask` is ignored for such a barrier, such that no visibility operation is executed - the value of this mask does not affect the validity of the barrier. The release operation happens-after the availability operation, and happens-before operations specified in the second synchronization scope of the calling command.

An *acquire operation* is used to acquire exclusive ownership of a range of a buffer or image subresource range. An acquire operation is defined by executing a [buffer memory barrier](#) (for a buffer range) or an [image memory barrier](#) (for an image subresource range) using `vkCmdPipelineBarrier`, on a queue from the destination queue family. The buffer range or image subresource range specified in an acquire operation **must** match exactly that of a previous release operation. The `srcQueueFamilyIndex` parameter of the barrier **must** be set to the source queue family index, and the `dstQueueFamilyIndex` parameter to the destination queue family index. `srcAccessMask` is ignored for such a barrier, such that no availability operation is executed - the value of this mask does not affect the validity of the barrier. The acquire operation happens-after operations in the first synchronization scope of the calling command, and happens-before the visibility operation.

*Note*



Whilst it is not invalid to provide destination or source access masks for memory barriers used for release or acquire operations, respectively, they have no practical effect. Access after a release operation has undefined results, and so visibility for those accesses has no practical effect. Similarly, write access before an acquire operation will produce undefined results for future access, so availability of those writes has no practical use. In an earlier version of the specification, these were required to match on both sides - but this was subsequently relaxed. These masks **should** be set to 0.

If the transfer is via an image memory barrier, and an [image layout transition](#) is desired, then the values of `oldLayout` and `newLayout` in the release memory barrier **must** be equal to values of `oldLayout` and `newLayout` in the acquire memory barrier. Although the image layout transition is submitted twice, it will only be executed once. A layout transition specified in this way happens-after the release operation and happens-before the acquire operation.

If the values of `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are equal, no ownership transfer is performed, and the barrier operates as if they were both set to `VK_QUEUE_FAMILY_IGNORED`.

Queue family ownership transfers **may** perform read and write accesses on all memory bound to the image subresource or buffer range, so applications **must** ensure that all memory writes have been made [available](#) before a queue family ownership transfer is executed. Available memory is automatically made visible to queue family release and acquire operations, and writes performed by those operations are automatically made available.

Once a queue family has acquired ownership of a buffer range or image subresource range of a `VK_SHARING_MODE_EXCLUSIVE` resource, its contents are undefined to other queue families unless ownership is transferred. The contents of any portion of another resource which aliases memory that is bound to the transferred buffer or image subresource range are undefined after a release or acquire operation.

*Note*

Because `events` **cannot** be used directly for inter-queue synchronization, and because `vkCmdSetEvent` does not have the queue family index or memory barrier parameters needed by a *release operation*, the release and acquire operations of a queue family ownership transfer **can** only be performed using `vkCmdPipelineBarrier`.



## 6.8. Wait Idle Operations

To wait on the host for the completion of outstanding queue operations for a given queue, call:

```
VkResult vkQueueWaitIdle(  
    VkQueue  
        queue);
```

- `queue` is the queue on which to wait.

`vkQueueWaitIdle` is equivalent to submitting a fence to a queue and waiting with an infinite timeout for that fence to signal.

### Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle

### Host Synchronization

- Host access to `queue` **must** be externally synchronized

### Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	Any	-

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

To wait on the host for the completion of outstanding queue operations for all queues on a given logical device, call:

```
VkResult vkDeviceWaitIdle(  
    VkDevice                device);
```

- `device` is the logical device to idle.

`vkDeviceWaitIdle` is equivalent to calling `vkQueueWaitIdle` for all queues owned by `device`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle

## Host Synchronization

- Host access to all `VkQueue` objects created from `device` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

## 6.9. Host Write Ordering Guarantees

When batches of command buffers are submitted to a queue via `vkQueueSubmit`, it defines a memory dependency with prior host operations, and execution of command buffers submitted to

the queue.

The first [synchronization scope](#) is defined by the host execution model, but includes execution of `vkQueueSubmit` on the host and anything that happened-before it.

The second [synchronization scope](#) includes all commands submitted in the same [queue submission](#), and all commands that occur later in [submission order](#).

The first [access scope](#) includes all host writes to mappable device memory that are available to the host memory domain.

The second [access scope](#) includes all memory access performed by the device.

## 6.10. Synchronization and Multiple Physical Devices

If a logical device includes more than one physical device, then fences, semaphores, and events all still have a single instance of the signaled state.

A fence becomes signaled when all physical devices complete the necessary queue operations.

Semaphore wait and signal operations all include a device index that is the sole physical device that performs the operation. These indices are provided in the [VkDeviceGroupSubmitInfo](#) and [VkDeviceGroupBindSparseInfo](#) structures. Semaphores are not exclusively owned by any physical device. For example, a semaphore can be signaled by one physical device and then waited on by a different physical device.

An event **can** only be waited on by the same physical device that signaled it (or the host).

## 6.11. Calibrated timestamps

In order to be able to correlate the time a particular operation took place at on timelines of different time domains (e.g. a device operation vs a host operation), Vulkan allows querying calibrated timestamps from multiple time domains.

To query calibrated timestamps from a set of time domains, call:

```
VkResult vkGetCalibratedTimestampsEXT(  
    VkDevice                                     device,  
    uint32_t                                      timestampCount,  
    const VkCalibratedTimestampInfoEXT*          pTimestampInfos,  
    uint64_t*                                     pTimestamps,  
    uint64_t*                                     pMaxDeviation);
```

- `device` is the logical device used to perform the query.
- `timestampCount` is the number of timestamps to query.
- `pTimestampInfos` is a pointer to an array of `timestampCount` [VkCalibratedTimestampInfoEXT](#) structures, describing the time domains the calibrated timestamps should be captured from.

- `pTimestamps` is a pointer to an array of `timestampCount` 64-bit unsigned integer values in which the requested calibrated timestamp values are returned.
- `pMaxDeviation` is a pointer to a 64-bit unsigned integer value in which the strictly positive maximum deviation, in nanoseconds, of the calibrated timestamp values is returned.

**Note**

The maximum deviation **may** vary between calls to `vkGetCalibratedTimestampsEXT` even for the same set of time domains due to implementation and platform specific reasons. It is the application's responsibility to assess whether the returned maximum deviation makes the timestamp values suitable for any particular purpose and **can** choose to re-issue the timestamp calibration call pursuing a lower deviation value.



Calibrated timestamp values **can** be extrapolated to estimate future coinciding timestamp values, however, depending on the nature of the time domains and other properties of the platform extrapolating values over a sufficiently long period of time **may** no longer be accurate enough to fit any particular purpose so applications are expected to re-calibrate the timestamps on a regular basis.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pTimestampInfos` **must** be a valid pointer to an array of `timestampCount` valid `VkCalibratedTimestampInfoEXT` structures
- `pTimestamps` **must** be a valid pointer to an array of `timestampCount uint64_t` values
- `pMaxDeviation` **must** be a valid pointer to a `uint64_t` value
- `timestampCount` **must** be greater than 0

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkCalibratedTimestampInfoEXT` structure is defined as:

```
typedef struct VkCalibratedTimestampInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkTimeDomainEXT timeDomain;
} VkCalibratedTimestampInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `timeDomain` is a `VkTimeDomainEXT` value specifying the time domain from which the calibrated timestamp value should be returned.

## Valid Usage

- `timeDomain` **must** be one of the `VkTimeDomainEXT` values returned by `vkGetPhysicalDeviceCalibrateableTimeDomainsEXT`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT`
- `pNext` **must** be `NULL`
- `timeDomain` **must** be a valid `VkTimeDomainEXT` value

The set of supported time domains consists of:

```
typedef enum VkTimeDomainEXT {
    VK_TIME_DOMAIN_DEVICE_EXT = 0,
    VK_TIME_DOMAIN_CLOCK_MONOTONIC_EXT = 1,
    VK_TIME_DOMAIN_CLOCK_MONOTONIC_RAW_EXT = 2,
    VK_TIME_DOMAIN_QUERY_PERFORMANCE_COUNTER_EXT = 3,
    VK_TIME_DOMAIN_MAX_ENUM_EXT = 0x7FFFFFFF
} VkTimeDomainEXT;
```

- `VK_TIME_DOMAIN_DEVICE_EXT` specifies the device time domain. Timestamp values in this time domain use the same units and are comparable with device timestamp values captured using `vkCmdWriteTimestamp` and are defined to be incrementing according to the `timestampPeriod` of the device.
- `VK_TIME_DOMAIN_CLOCK_MONOTONIC_EXT` specifies the `CLOCK_MONOTONIC` time domain available on POSIX platforms. Timestamp values in this time domain are in units of nanoseconds and are comparable with platform timestamp values captured using the POSIX `clock_gettime` API as computed by this example:

```
struct timespec tv;
clock_gettime(CLOCK_MONOTONIC, &tv);
return tv.tv_nsec + tv.tv_sec*1000000000ull;
```

- `VK_TIME_DOMAIN_CLOCK_MONOTONIC_RAW_EXT` specifies the `CLOCK_MONOTONIC_RAW` time domain available on POSIX platforms. Timestamp values in this time domain are in units of nanoseconds and are comparable with platform timestamp values captured using the POSIX `clock_gettime` API as computed by this example:

```
struct timespec tv;
clock_gettime(CLOCK_MONOTONIC_RAW, &tv);
return tv.tv_nsec + tv.tv_sec*1000000000ull;
```

- `VK_TIME_DOMAIN_QUERY_PERFORMANCE_COUNTER_EXT` specifies the performance counter (QPC) time domain available on Windows. Timestamp values in this time domain are in the same units as those provided by the Windows `QueryPerformanceCounter` API and are comparable with platform timestamp values captured using that API as computed by this example:

```
LARGE_INTEGER counter;
QueryPerformanceCounter(&counter);
return counter.QuadPart;
```

# Chapter 7. Render Pass

A *render pass* represents a collection of attachments, subpasses, and dependencies between the subpasses, and describes how the attachments are used over the course of the subpasses. The use of a render pass in a command buffer is a *render pass instance*.

Render passes are represented by `VkRenderPass` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkRenderPass)
```

An *attachment description* describes the properties of an attachment including its format, sample count, and how its contents are treated at the beginning and end of each render pass instance.

A *subpass* represents a phase of rendering that reads and writes a subset of the attachments in a render pass. Rendering commands are recorded into a particular subpass of a render pass instance.

A *subpass description* describes the subset of attachments that is involved in the execution of a subpass. Each subpass **can** read from some attachments as *input attachments*, write to some as *color attachments* or *depth/stencil attachments*, and perform *multisample resolve operations* to *resolve attachments*. A subpass description **can** also include a set of *preserve attachments*, which are attachments that are not read or written by the subpass but whose contents **must** be preserved throughout the subpass.

A subpass *uses an attachment* if the attachment is a color, depth/stencil, resolve, depth/stencil resolve, or input attachment for that subpass (as determined by the `pColorAttachments`, `pDepthStencilAttachment`, `pResolveAttachments`, `VkSubpassDescriptionDepthStencilResolveKHR::pDepthStencilResolveAttachment`, and `pInputAttachments` members of `VkSubpassDescription`, respectively). A subpass does not use an attachment if that attachment is preserved by the subpass. The *first use of an attachment* is in the lowest numbered subpass that uses that attachment. Similarly, the *last use of an attachment* is in the highest numbered subpass that uses that attachment.

The subpasses in a render pass all render to the same dimensions, and fragments for pixel (x,y,layer) in one subpass **can** only read attachment contents written by previous subpasses at that same (x,y,layer) location.

## Note

By describing a complete set of subpasses in advance, render passes provide the implementation an opportunity to optimize the storage and transfer of attachment data between subpasses.



In practice, this means that subpasses with a simple framebuffer-space dependency **may** be merged into a single tiled rendering pass, keeping the attachment data on-chip for the duration of a render pass instance. However, it is also quite common for a render pass to only contain a single subpass.

*Subpass dependencies* describe [execution and memory dependencies](#) between subpasses.

A *subpass dependency chain* is a sequence of subpass dependencies in a render pass, where the source subpass of each subpass dependency (after the first) equals the destination subpass of the previous dependency.

Execution of subpasses **may** overlap or execute out of order with regards to other subpasses, unless otherwise enforced by an execution dependency. Each subpass only respects [submission order](#) for commands recorded in the same subpass, and the `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands that delimit the render pass - commands within other subpasses are not included. This affects most other [implicit ordering guarantees](#).

A render pass describes the structure of subpasses and attachments independent of any specific image views for the attachments. The specific image views that will be used for the attachments, and their dimensions, are specified in `VkFramebuffer` objects. Framebuffers are created with respect to a specific render pass that the framebuffer is compatible with (see [Render Pass Compatibility](#)). Collectively, a render pass and a framebuffer define the complete render target state for one or more subpasses as well as the algorithmic dependencies between the subpasses.

The various pipeline stages of the drawing commands for a given subpass **may** execute concurrently and/or out of order, both within and across drawing commands, whilst still respecting [pipeline order](#). However for a given (x,y,layer,sample) sample location, certain per-sample operations are performed in [rasterization order](#).

## 7.1. Render Pass Creation

To create a render pass, call:

```
VkResult vkCreateRenderPass(  
    VkDevice device,  
    const VkRenderPassCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkRenderPass* pRenderPass);
```

- `device` is the logical device that creates the render pass.
- `pCreateInfo` is a pointer to a `VkRenderPassCreateInfo` structure describing the parameters of the render pass.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pRenderPass` is a pointer to a `VkRenderPass` handle in which the resulting render pass object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkRenderPassCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pRenderPass` **must** be a valid pointer to a `VkRenderPass` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkRenderPassCreateInfo` structure is defined as:

```
typedef struct VkRenderPassCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkRenderPassCreateFlags     flags;
    uint32_t                  attachmentCount;
    const VkAttachmentDescription* pAttachments;
    uint32_t                  subpassCount;
    const VkSubpassDescription* pSubpasses;
    uint32_t                  dependencyCount;
    const VkSubpassDependency* pDependencies;
} VkRenderPassCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `attachmentCount` is the number of attachments used by this render pass.
- `pAttachments` is a pointer to an array of `attachmentCount` `VkAttachmentDescription` structures describing the attachments used by the render pass.
- `subpassCount` is the number of subpasses to create.
- `pSubpasses` is a pointer to an array of `subpassCount` `VkSubpassDescription` structures describing each subpass.
- `dependencyCount` is the number of memory dependencies between pairs of subpasses.

- `pDependencies` is a pointer to an array of `dependencyCount` `VkSubpassDependency` structures describing dependencies between pairs of subpasses.

*Note*



Care should be taken to avoid a data race here; if any subpasses access attachments with overlapping memory locations, and one of those accesses is a write, a subpass dependency needs to be included between them.

## Valid Usage

- If the `attachment` member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment`, or any element of `pPreserveAttachments` in any element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, it **must** be less than `attachmentCount`
- For any member of `pAttachments` with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`.
- For any member of `pAttachments` with a `stencilLoadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`.
- For any member of `pAttachments` with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.
- For any member of `pAttachments` with a `stencilLoadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`.
- If the `pNext` chain includes an instance of `VkRenderPassInputAttachmentAspectCreateInfo`, the `subpass` member of each element of its `pAspectReferences` member **must** be less than `subpassCount`
- If the `pNext` chain includes an instance of `VkRenderPassInputAttachmentAspectCreateInfo`, the `inputAttachmentIndex` member of each element of its `pAspectReferences` member **must** be less than the value of `inputAttachmentCount` in the member of `pSubpasses` identified by its `subpass` member
- If the `pNext` chain includes an instance of `VkRenderPassInputAttachmentAspectCreateInfo`, for any element of the `pInputAttachments` member of any element of `pSubpasses` where the `attachment` member is not `VK_ATTACHMENT_UNUSED`, the `aspectMask` member of the corresponding element of `VkRenderPassInputAttachmentAspectCreateInfo::pAspectReferences` **must** only include aspects that are present in images of the format specified by the element of `pAttachments` at `attachment`
- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, and its `subpassCount` member is not zero, that member **must** be equal to the value of `subpassCount`
- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, if its `dependencyCount` member is not zero, it **must** be equal to `dependencyCount`
- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, for each non-zero element of `pViewOffsets`, the `srcSubpass` and `dstSubpass` members of `pDependencies` at the same index **must** not be equal
- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, for any element of `pDependencies` with a `dependencyFlags` member that does not include

`VK_DEPENDENCY_VIEW_LOCAL_BIT`, the corresponding element of the `pViewOffsets` member of that `VkRenderPassMultiviewCreateInfo` instance **must** be `0`

- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, elements of its `pViewMasks` member **must** either all be `0`, or all not be `0`
- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, and each element of its `pViewMasks` member is `0`, the `dependencyFlags` member of each element of `pDependencies` **must** not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`
- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, and each element of its `pViewMasks` member is `0`, `correlatedViewMaskCount` **must** be `0`
- If the `pNext` chain includes an instance of `VkRenderPassMultiviewCreateInfo`, each element of its `pViewMask` member **must** not have a bit set at an index greater than or equal to `VkPhysicalDeviceLimits::maxFramebufferLayers`
- For any element of `pDependencies`, if the `srcSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `srcStageMask` member of that dependency **must** be a pipeline stage supported by the `pipeline` identified by the `pipelineBindPoint` member of the source subpass
- For any element of `pDependencies`, if the `dstSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `dstStageMask` member of that dependency **must** be a pipeline stage supported by the `pipeline` identified by the `pipelineBindPoint` member of the destination subpass
- The `srcSubpass` member of each element of `pDependencies` **must** be less than `subpassCount`
- The `dstSubpass` member of each element of `pDependencies` **must** be less than `subpassCount`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkRenderPassFragmentDensityMapCreateInfoEXT`, `VkRenderPassInputAttachmentAspectCreateInfo`, or `VkRenderPassMultiviewCreateInfo`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be `0`
- If `attachmentCount` is not `0`, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkAttachmentDescription` structures
- `pSubpasses` **must** be a valid pointer to an array of `subpassCount` valid `VkSubpassDescription` structures
- If `dependencyCount` is not `0`, `pDependencies` **must** be a valid pointer to an array of `dependencyCount` valid `VkSubpassDependency` structures
- `subpassCount` **must** be greater than `0`

```
typedef VkFlags VkRenderPassCreateFlags;
```

`VkRenderPassCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

If the `VkRenderPassCreateInfo::pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure, then that structure includes an array of view masks, view offsets, and correlation masks for the render pass.

The `VkRenderPassMultiviewCreateInfo` structure is defined as:

```
typedef struct VkRenderPassMultiviewCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t subpassCount;
    const uint32_t* pViewMasks;
    uint32_t dependencyCount;
    const int32_t* pViewOffsets;
    uint32_t correlationMaskCount;
    const uint32_t* pCorrelationMasks;
} VkRenderPassMultiviewCreateInfo;
```

or the equivalent

```
typedef VkRenderPassMultiviewCreateInfo VkRenderPassMultiviewCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `subpassCount` is zero or the number of subpasses in the render pass.
- `pViewMasks` is a pointer to an array of `subpassCount` view masks, where each mask is a bitfield of view indices describing which views rendering is broadcast to in each subpass, when multiview is enabled. If `subpassCount` is zero, each view mask is treated as zero.
- `dependencyCount` is zero or the number of dependencies in the render pass.
- `pViewOffsets` is a pointer to an array of `dependencyCount` view offsets, one for each dependency. If `dependencyCount` is zero, each dependency's view offset is treated as zero. Each view offset controls which views in the source subpass the views in the destination subpass depend on.
- `correlationMaskCount` is zero or the number of correlation masks.
- `pCorrelationMasks` is a pointer to an array of `correlationMaskCount` view masks indicating sets of views that **may** be more efficient to render concurrently.

When a subpass uses a non-zero view mask, *multiview* functionality is considered to be enabled. Multiview is all-or-nothing for a render pass - that is, either all subpasses **must** have a non-zero view mask (though some subpasses **may** have only one view) or all **must** be zero. Multiview causes

all drawing and clear commands in the subpass to behave as if they were broadcast to each view, where a view is represented by one layer of the framebuffer attachments. All draws and clears are broadcast to each *view index* whose bit is set in the view mask. The view index is provided in the `ViewIndex` shader input variable, and color, depth/stencil, and input attachments all read/write the layer of the framebuffer corresponding to the view index.

If the view mask is zero for all subpasses, multiview is considered to be disabled and all drawing commands execute normally, without this additional broadcasting.

Some implementations **may** not support multiview in conjunction with [geometry shaders](#) or [tessellation shaders](#).

When multiview is enabled, the `VK_DEPENDENCY_VIEW_LOCAL_BIT` bit in a dependency **can** be used to express a view-local dependency, meaning that each view in the destination subpass depends on a single view in the source subpass. Unlike pipeline barriers, a subpass dependency **can** potentially have a different view mask in the source subpass and the destination subpass. If the dependency is view-local, then each view (`dstView`) in the destination subpass depends on the view `dstView + pViewOffsets[dependency]` in the source subpass. If there is not such a view in the source subpass, then this dependency does not affect that view in the destination subpass. If the dependency is not view-local, then all views in the destination subpass depend on all views in the source subpass, and the view offset is ignored. A non-zero view offset is not allowed in a self-dependency.

The elements of `pCorrelationMasks` are a set of masks of views indicating that views in the same mask **may** exhibit spatial coherency between the views, making it more efficient to render them concurrently. Correlation masks **must** not have a functional effect on the results of the multiview rendering.

When multiview is enabled, at the beginning of each subpass all non-render pass state is undefined. In particular, each time `vkCmdBeginRenderPass` or `vkCmdNextSubpass` is called the graphics pipeline **must** be bound, any relevant descriptor sets or vertex/index buffers **must** be bound, and any relevant dynamic state or push constants **must** be set before they are used.

A multiview subpass **can** declare that its shaders will write per-view attributes for all views in a single invocation, by setting the `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX` bit in the subpass description. The only supported per-view attributes are position and viewport mask, and per-view position and viewport masks are written to output array variables decorated with `PositionPerViewNV` and `ViewportMaskPerViewNV`, respectively. If `VK_NV_viewport_array2` is not supported and enabled, `ViewportMaskPerViewNV` **must** not be used. Values written to elements of `PositionPerViewNV` and `ViewportMaskPerViewNV` **must** not depend on the `ViewIndex`. The shader **must** also write to an output variable decorated with `Position`, and the value written to `Position` **must** equal the value written to `PositionPerViewNV[ViewIndex]`. Similarly, if `ViewportMaskPerViewNV` is written to then the shader **must** also write to an output variable decorated with `ViewportMaskNV`, and the value written to `ViewportMaskNV` **must** equal the value written to `ViewportMaskPerViewNV[ViewIndex]`. Implementations will either use values taken from `Position` and `ViewportMaskNV` and invoke the shader once for each view, or will use values taken from `PositionPerViewNV` and `ViewportMaskPerViewNV` and invoke the shader fewer times. The values written to `Position` and `ViewportMaskNV` **must** not depend on the values written to `PositionPerViewNV` and `ViewportMaskPerViewNV`, or vice versa (to allow compilers to eliminate the unused outputs). All attributes that do not have `*PerViewNV` counterparts **must** not depend on `ViewIndex`.

Per-view attributes are all-or-nothing for a subpass. That is, all pipelines compiled against a subpass that includes the `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX` bit **must** write per-view attributes to the `*PerViewNV[]` shader outputs, in addition to the non-per-view (e.g. `Position`) outputs. Pipelines compiled against a subpass that does not include this bit **must** not include the `*PerViewNV[]` outputs in their interfaces.

## Valid Usage

- Each view index **must** not be set in more than one element of `pCorrelationMasks`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO`
- If `subpassCount` is not `0`, `pViewMasks` **must** be a valid pointer to an array of `subpassCount uint32_t` values
- If `dependencyCount` is not `0`, `pViewOffsets` **must** be a valid pointer to an array of `dependencyCount int32_t` values
- If `correlationMaskCount` is not `0`, `pCorrelationMasks` **must** be a valid pointer to an array of `correlationMaskCount uint32_t` values

If the `VkRenderPassCreateInfo::pNext` chain includes a `VkRenderPassFragmentDensityMapCreateInfoEXT` structure, then that structure includes a fragment density map attachment for the render pass.

The `VkRenderPassFragmentDensityMapCreateInfoEXT` structure is defined as:

```
typedef struct VkRenderPassFragmentDensityMapCreateInfoEXT {
    VkStructureType          sType;
    const void*               pNext;
    VkAttachmentReference     fragmentDensityMapAttachment;
} VkRenderPassFragmentDensityMapCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `fragmentDensityMapAttachment` is the fragment density map to use for the render pass.

The fragment density map attachment is read at an implementation-dependent time either by the host during `vkCmdBeginRenderPass` if the attachment's image view was not created with `flags` containing `VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT`, or by the device when drawing commands in the renderpass execute `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`.

If this structure is not present, it is as if `fragmentDensityMapAttachment` was given as `VK_ATTACHMENT_UNUSED`.

## Valid Usage

- If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` **must** be less than `VkRenderPassCreateInfo::attachmentCount`
- If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` **must** not be an element of `VkSubpassDescription::pInputAttachments`, `VkSubpassDescription::pColorAttachments`, `VkSubpassDescription::pResolveAttachments`, `VkSubpassDescription::pDepthStencilAttachment`, or `VkSubpassDescription::pPreserveAttachments` for any subpass
- If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `layout` **must** be equal to `VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT`, or `VK_IMAGE_LAYOUT_GENERAL`
- If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` **must** reference an attachment with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_LOAD` or `VK_ATTACHMENT_LOAD_OP_DONT_CARE`.
- If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` **must** reference an attachment with a `storeOp` equal to `VK_ATTACHMENT_STORE_OP_DONT_CARE`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_FRAGMENT_DENSITY_MAP_CREATE_INFO_EXT`
- `fragmentDensityMapAttachment` **must** be a valid `VkAttachmentReference` structure

The `VkAttachmentDescription` structure is defined as:

```
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                      format;
    VkSampleCountFlagBits         samples;
    VkAttachmentLoadOp            loadOp;
    VkAttachmentStoreOp           storeOp;
    VkAttachmentLoadOp            stencilLoadOp;
    VkAttachmentStoreOp           stencilStoreOp;
    VkImageLayout                  initialLayout;
    VkImageLayout                  finalLayout;
} VkAttachmentDescription;
```

- `flags` is a bitmask of `VkAttachmentDescriptionFlagBits` specifying additional properties of the attachment.
- `format` is a `VkFormat` value specifying the format of the image view that will be used for the attachment.
- `samples` is the number of samples of the image as defined in `VkSampleCountFlagBits`.
- `loadOp` is a `VkAttachmentLoadOp` value specifying how the contents of color and depth

components of the attachment are treated at the beginning of the subpass where it is first used.

- `storeOp` is a `VkAttachmentStoreOp` value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
- `stencilLoadOp` is a `VkAttachmentLoadOp` value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.
- `stencilStoreOp` is a `VkAttachmentStoreOp` value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.
- `initialLayout` is the layout the attachment image subresource will be in when a render pass instance begins.
- `finalLayout` is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

If the attachment uses a color format, then `loadOp` and `storeOp` are used, and `stencilLoadOp` and `stencilStoreOp` are ignored. If the format has depth and/or stencil components, `loadOp` and `storeOp` apply only to the depth data, while `stencilLoadOp` and `stencilStoreOp` define how the stencil data is handled. `loadOp` and `stencilLoadOp` define the *load operations* that execute as part of the first subpass that uses the attachment. `storeOp` and `stencilStoreOp` define the *store operations* that execute as part of the last subpass that uses the attachment.

The load operation for each sample in an attachment happens-before any recorded command which accesses the sample in the first subpass where the attachment is used. Load operations for attachments with a depth/stencil format execute in the `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT` pipeline stage. Load operations for attachments with a color format execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage.

The store operation for each sample in an attachment happens-after any recorded command which accesses the sample in the last subpass where the attachment is used. Store operations for attachments with a depth/stencil format execute in the `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT` pipeline stage. Store operations for attachments with a color format execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage.

If an attachment is not used by any subpass, then `loadOp`, `storeOp`, `stencilStoreOp`, and `stencilLoadOp` are ignored, and the attachment's memory contents will not be modified by execution of a render pass instance.

The load and store operations apply on the first and last use of each view in the render pass, respectively. If a view index of an attachment is not included in the view mask in any subpass that uses it, then the load and store operations are ignored, and the attachment's memory contents will not be modified by execution of a render pass instance.

During a render pass instance, input/color attachments with color formats that have a component size of 8, 16, or 32 bits **must** be represented in the attachment's format throughout the instance. Attachments with other floating- or fixed-point color formats, or with depth components **may** be represented in a format with a precision higher than the attachment format, but **must** be represented with the same range. When such a component is loaded via the `loadOp`, it will be converted into an implementation-dependent format used by the render pass. Such components **must** be converted from the render pass format, to the format of the attachment, before they are

resolved or stored at the end of a render pass instance via `storeOp`. Conversions occur as described in [Numeric Representation and Computation](#) and [Fixed-Point Data Conversions](#).

If `flags` includes `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT`, then the attachment is treated as if it shares physical memory with another attachment in the same render pass. This information limits the ability of the implementation to reorder certain operations (like layout transitions and the `loadOp`) such that it is not improperly reordered against other uses of the same physical memory via a different attachment. This is described in more detail below.

## Valid Usage

- `finalLayout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`
- If the `separateDepthStencilLayouts` feature is not enabled, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` or
- If the `separateDepthStencilLayouts` feature is not enabled, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` or
- If `format` is a color format, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` or
- If `format` is a color format, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` or
- If `format` is a depth/stencil format which includes both depth and stencil aspects, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` or
- If `format` is a depth/stencil format which includes both depth and stencil aspects, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` or
- If `format` is a depth/stencil format which includes only the depth aspect, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If `format` is a depth/stencil format which includes only the depth aspect, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If `format` is a depth/stencil format which includes only the stencil aspect, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`
- If `format` is a depth/stencil format which includes only the stencil aspect, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`

## Valid Usage (Implicit)

- **flags** **must** be a valid combination of [VkAttachmentDescriptionFlagBits](#) values
- **format** **must** be a valid [VkFormat](#) value
- **samples** **must** be a valid [VkSampleCountFlagBits](#) value
- **loadOp** **must** be a valid [VkAttachmentLoadOp](#) value
- **storeOp** **must** be a valid [VkAttachmentStoreOp](#) value
- **stencilLoadOp** **must** be a valid [VkAttachmentLoadOp](#) value
- **stencilStoreOp** **must** be a valid [VkAttachmentStoreOp](#) value
- **initialLayout** **must** be a valid [VkImageLayout](#) value
- **finalLayout** **must** be a valid [VkImageLayout](#) value

To specify which aspects of an input attachment **can** be read add a [VkRenderPassInputAttachmentAspectCreateInfo](#) structure to the **pNext** chain of the [VkRenderPassCreateInfo](#) structure:

The [VkRenderPassInputAttachmentAspectCreateInfo](#) structure is defined as:

```
typedef struct VkRenderPassInputAttachmentAspectCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    uint32_t                  aspectReferenceCount;
    const VkInputAttachmentAspectReference* pAspectReferences;
} VkRenderPassInputAttachmentAspectCreateInfo;
```

or the equivalent

```
typedef VkRenderPassInputAttachmentAspectCreateInfo
VkRenderPassInputAttachmentAspectCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **aspectReferenceCount** is the number of elements in the **pAspectReferences** array.
- **pAspectReferences** is a pointer to an array of **aspectReferenceCount** [VkInputAttachmentAspectReference](#) structures describing which aspect(s) **can** be accessed for a given input attachment within a given subpass.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO`
- `pAspectReferences` **must** be a valid pointer to an array of `aspectReferenceCount` valid `VkInputAttachmentAspectReference` structures
- `aspectReferenceCount` **must** be greater than `0`

The `VkInputAttachmentAspectReference` structure specifies an aspect mask for a specific input attachment of a specific subpass in the render pass.

`subpass` and `inputAttachmentIndex` index into the render pass as:

```
pCreateInfo::pSubpasses[subpass].pInputAttachments[inputAttachmentIndex]
```

```
typedef struct VkInputAttachmentAspectReference {
    uint32_t          subpass;
    uint32_t          inputAttachmentIndex;
    VkImageAspectFlags aspectMask;
} VkInputAttachmentAspectReference;
```

or the equivalent

```
typedef VkInputAttachmentAspectReference VkInputAttachmentAspectReferenceKHR;
```

- `subpass` is an index into the `pSubpasses` array of the parent `VkRenderPassCreateInfo` structure.
- `inputAttachmentIndex` is an index into the `pInputAttachments` of the specified subpass.
- `aspectMask` is a mask of which aspect(s) **can** be accessed within the specified subpass.

## Valid Usage

- `aspectMask` **must** not include `VK_IMAGE_ASPECT_METADATA_BIT`
- `aspectMask` **must** not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`.

## Valid Usage (Implicit)

- `aspectMask` **must** be a valid combination of `VkImageAspectFlagBits` values
- `aspectMask` **must** not be `0`

An application **must** only access the specified aspect(s).

An application **can** access any aspect of an input attachment that does not have a specified aspect mask.

Bits which **can** be set in `VkAttachmentDescription::flags` describing additional properties of the attachment are:

```
typedef enum VkAttachmentDescriptionFlagBits {
    VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT = 0x00000001,
    VK_ATTACHMENT_DESCRIPTION_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkAttachmentDescriptionFlagBits;
```

- `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` specifies that the attachment aliases the same device memory as other attachments.

```
typedef VkFlags VkAttachmentDescriptionFlags;
```

`VkAttachmentDescriptionFlags` is a bitmask type for setting a mask of zero or more `VkAttachmentDescriptionFlagBits`.

Possible values of `VkAttachmentDescription::loadOp` and `stencilLoadOp`, specifying how the contents of the attachment are treated, are:

```
typedef enum VkAttachmentLoadOp {
    VK_ATTACHMENT_LOAD_OP_LOAD = 0,
    VK_ATTACHMENT_LOAD_OP_CLEAR = 1,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE = 2,
    VK_ATTACHMENT_LOAD_OP_MAX_ENUM = 0x7FFFFFFF
} VkAttachmentLoadOp;
```

- `VK_ATTACHMENT_LOAD_OP_LOAD` specifies that the previous contents of the image within the render area will be preserved. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`.
- `VK_ATTACHMENT_LOAD_OP_CLEAR` specifies that the contents within the render area will be cleared to a uniform value, which is specified when a render pass instance is begun. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.
- `VK_ATTACHMENT_LOAD_OP_DONT_CARE` specifies that the previous contents within the area need not be preserved; the contents of the attachment will be undefined inside the render area. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

Possible values of `VkAttachmentDescription::storeOp` and `stencilStoreOp`, specifying how the contents of the attachment are treated, are:

```

typedef enum VkAttachmentStoreOp {
    VK_ATTACHMENT_STORE_OP_STORE = 0,
    VK_ATTACHMENT_STORE_OP_DONT_CARE = 1,
    VK_ATTACHMENT_STORE_OP_MAX_ENUM = 0x7FFFFFFF
} VkAttachmentStoreOp;

```

- `VK_ATTACHMENT_STORE_OP_STORE` specifies the contents generated during the render pass and within the render area are written to memory. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.
- `VK_ATTACHMENT_STORE_OP_DONT_CARE` specifies the contents within the render area are not needed after rendering, and **may** be discarded; the contents of the attachment will be undefined inside the render area. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

If a render pass uses multiple attachments that alias the same device memory, those attachments **must** each include the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` bit in their attachment description flags. Attachments aliasing the same memory occurs in multiple ways:

- Multiple attachments being assigned the same image view as part of framebuffer creation.
- Attachments using distinct image views that correspond to the same image subresource of an image.
- Attachments using views of distinct image subresources which are bound to overlapping memory ranges.

*Note*

Render passes **must** include subpass dependencies (either directly or via a subpass dependency chain) between any two subpasses that operate on the same attachment or aliasing attachments and those subpass dependencies **must** include execution and memory dependencies separating uses of the aliases, if at least one of those subpasses writes to one of the aliases. These dependencies **must** not include the `VK_DEPENDENCY_BY_REGION_BIT` if the aliases are views of distinct image subresources which overlap in memory.



Multiple attachments that alias the same memory **must** not be used in a single subpass. A given attachment index **must** not be used multiple times in a single subpass, with one exception: two subpass attachments **can** use the same attachment index if at least one use is as an input attachment and neither use is as a resolve or preserve attachment. In other words, the same view **can** be used simultaneously as an input and color or depth/stencil attachment, but **must** not be used as multiple color or depth/stencil attachments nor as resolve or preserve attachments. The precise set of valid scenarios is described in more detail [below](#).

If a set of attachments alias each other, then all except the first to be used in the render pass **must** use an `initialLayout` of `VK_IMAGE_LAYOUT_UNDEFINED`, since the earlier uses of the other aliases make their contents undefined. Once an alias has been used and a different alias has been used after it,

the first alias **must** not be used in any later subpasses. However, an application **can** assign the same image view to multiple aliasing attachment indices, which allows that image view to be used multiple times even if other aliases are used in between.

*Note*



Once an attachment needs the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` bit, there **should** be no additional cost of introducing additional aliases, and using these additional aliases **may** allow more efficient clearing of the attachments on multiple uses via `VK_ATTACHMENT_LOAD_OP_CLEAR`.

The `VkSubpassDescription` structure is defined as:

```
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags      flags;
    VkPipelineBindPoint           pipelineBindPoint;
    uint32_t                      inputAttachmentCount;
    const VkAttachmentReference* pInputAttachments;
    uint32_t                      colorAttachmentCount;
    const VkAttachmentReference* pColorAttachments;
    const VkAttachmentReference* pResolveAttachments;
    const VkAttachmentReference* pDepthStencilAttachment;
    uint32_t                      preserveAttachmentCount;
    const uint32_t*                pPreserveAttachments;
} VkSubpassDescription;
```

- `flags` is a bitmask of `VkSubpassDescriptionFlagBits` specifying usage of the subpass.
- `pipelineBindPoint` is a `VkPipelineBindPoint` value specifying the pipeline type supported for this subpass.
- `inputAttachmentCount` is the number of input attachments.
- `pInputAttachments` is a pointer to an array of `VkAttachmentReference` structures defining the input attachments for this subpass and their layouts.
- `colorAttachmentCount` is the number of color attachments.
- `pColorAttachments` is a pointer to an array of `VkAttachmentReference` structures defining the color attachments for this subpass and their layouts.
- `pResolveAttachments` is an optional array of `colorAttachmentCount` `VkAttachmentReference` structures defining the resolve attachments for this subpass and their layouts.
- `pDepthStencilAttachment` is a pointer to a `VkAttachmentReference` structure specifying the depth/stencil attachment for this subpass and its layout.
- `preserveAttachmentCount` is the number of preserved attachments.
- `pPreserveAttachments` is a pointer to an array of `preserveAttachmentCount` render pass attachment indices identifying attachments that are not used by this subpass, but whose contents **must** be preserved throughout the subpass.

Each element of the `pInputAttachments` array corresponds to an input attachment index in a

fragment shader, i.e. if a shader declares an image variable decorated with a `InputAttachmentIndex` value of `X`, then it uses the attachment provided in `pInputAttachments[X]`. Input attachments **must** also be bound to the pipeline in a descriptor set. If the `attachment` member of any element of `pInputAttachments` is `VK_ATTACHMENT_UNUSED`, the application **must** not read from the corresponding input attachment index. Fragment shaders **can** use subpass input variables to access the contents of an input attachment at the fragment's (x, y, layer) framebuffer coordinates.

Each element of the `pColorAttachments` array corresponds to an output location in the shader, i.e. if the shader declares an output variable decorated with a `Location` value of `X`, then it uses the attachment provided in `pColorAttachments[X]`. If the `attachment` member of any element of `pColorAttachments` is `VK_ATTACHMENT_UNUSED`, writes to the corresponding location by a fragment are discarded.

If `pResolveAttachments` is not `NULL`, each of its elements corresponds to a color attachment (the element in `pColorAttachments` at the same index), and a multisample resolve operation is defined for each attachment. At the end of each subpass, multisample resolve operations read the subpass's color attachments, and resolve the samples for each pixel within the render area to the same pixel location in the corresponding resolve attachments, unless the resolve attachment index is `VK_ATTACHMENT_UNUSED`.

Similarly, if `VkSubpassDescriptionDepthStencilResolveKHR::pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, it corresponds to the depth/stencil attachment in `pDepthStencilAttachment`, and multisample resolve operations for depth and stencil are defined by `VkSubpassDescriptionDepthStencilResolveKHR::depthResolveMode` and `VkSubpassDescriptionDepthStencilResolveKHR::stencilResolveMode`, respectively. At the end of each subpass, multisample resolve operations read the subpass's depth/stencil attachment, and resolve the samples for each pixel to the same pixel location in the corresponding resolve attachment. If `VkSubpassDescriptionDepthStencilResolveKHR::depthResolveMode` is `VK_RESOLVE_MODE_NONE_KHR`, then the depth component of the resolve attachment is not written to and its contents are preserved. Similarly, if `VkSubpassDescriptionDepthStencilResolveKHR::stencilResolveMode` is `VK_RESOLVE_MODE_NONE_KHR`, then the stencil component of the resolve attachment is not written to and its contents are preserved. `VkSubpassDescriptionDepthStencilResolveKHR::depthResolveMode` is ignored if the `VkFormat` of the `pDepthStencilResolveAttachment` does not have a depth component. Similarly, `VkSubpassDescriptionDepthStencilResolveKHR::stencilResolveMode` is ignored if the `VkFormat` of the `pDepthStencilResolveAttachment` does not have a stencil component.

If the image subresource range referenced by the depth/stencil attachment is created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT`, then the multisample resolve operation uses the sample locations state specified in the `sampleLocationsInfo` member of the element of the `VkRenderPassSampleLocationsBeginInfoEXT::pPostSubpassSampleLocations` for the subpass.

If `pDepthStencilAttachment` is `NULL`, or if its attachment index is `VK_ATTACHMENT_UNUSED`, it indicates that no depth/stencil attachment will be used in the subpass.

The contents of an attachment within the render area become undefined at the start of a subpass `S` if all of the following conditions are true:

- The attachment is used as a color, depth/stencil, or resolve attachment in any subpass in the

render pass.

- There is a subpass  $S_1$  that uses or preserves the attachment, and a subpass dependency from  $S_1$  to  $S$ .
- The attachment is not used or preserved in subpass  $S$ .

Once the contents of an attachment become undefined in subpass  $S$ , they remain undefined for subpasses in subpass dependency chains starting with subpass  $S$  until they are written again. However, they remain valid for subpasses in other subpass dependency chains starting with subpass  $S_1$  if those subpasses use or preserve the attachment.

## Valid Usage

- `pipelineBindPoint` **must** be `VK_PIPELINE_BIND_POINT_GRAPHICS`
- `colorAttachmentCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxColorAttachments`
- If the first use of an attachment in this render pass is as an input attachment, and the attachment is not also used as a color or depth/stencil attachment in the same subpass, then `loadOp` **must** not be `VK_ATTACHMENT_LOAD_OP_CLEAR`
- If `pResolveAttachments` is not `NULL`, for each resolve attachment that is not `VK_ATTACHMENT_UNUSED`, the corresponding color attachment **must** not be `VK_ATTACHMENT_UNUSED`
- If `pResolveAttachments` is not `NULL`, for each resolve attachment that is not `VK_ATTACHMENT_UNUSED`, the corresponding color attachment **must** not have a sample count of `VK_SAMPLE_COUNT_1_BIT`
- If `pResolveAttachments` is not `NULL`, each resolve attachment that is not `VK_ATTACHMENT_UNUSED` **must** have a sample count of `VK_SAMPLE_COUNT_1_BIT`
- If `pResolveAttachments` is not `NULL`, each resolve attachment that is not `VK_ATTACHMENT_UNUSED` **must** have the same `VkFormat` as its corresponding color attachment
- All attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` **must** have the same sample count
- All attachments in `pInputAttachments` that are not `VK_ATTACHMENT_UNUSED` **must** have formats whose features contain at least one of `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` or `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.
- All attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` **must** have formats whose features contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`
- All attachments in `pResolveAttachments` that are not `VK_ATTACHMENT_UNUSED` **must** have formats whose features contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`
- If `pDepthStencilAttachment` is not `NULL` and the attachment is not `VK_ATTACHMENT_UNUSED` then it **must** have a format whose features contain `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If the `VK_AMD_mixed_attachment_samples` extension is enabled, and all attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` **must** have a sample count that is smaller than or equal to the sample count of `pDepthStencilAttachment` if it is not `VK_ATTACHMENT_UNUSED`
- If neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, and if `pDepthStencilAttachment` is not `VK_ATTACHMENT_UNUSED` and any attachments in `pColorAttachments` are not `VK_ATTACHMENT_UNUSED`, they **must** have the same sample count
- The `attachment` member of each element of `pPreserveAttachments` **must** not be `VK_ATTACHMENT_UNUSED`
- Each element of `pPreserveAttachments` **must** not also be an element of any other member

of the subpass description

- If any attachment is used by more than one `VkAttachmentReference` member, then each use **must** use the same `layout`
- If `flags` includes `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX`, it **must** also include `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX`.

## Valid Usage (Implicit)

- `flags` **must** be a valid combination of `VkSubpassDescriptionFlagBits` values
- `pipelineBindPoint` **must** be a valid `VkPipelineBindPoint` value
- If `inputAttachmentCount` is not `0`, `pInputAttachments` **must** be a valid pointer to an array of `inputAttachmentCount` valid `VkAttachmentReference` structures
- If `colorAttachmentCount` is not `0`, `pColorAttachments` **must** be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures
- If `colorAttachmentCount` is not `0`, and `pResolveAttachments` is not `NULL`, `pResolveAttachments` **must** be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures
- If `pDepthStencilAttachment` is not `NULL`, `pDepthStencilAttachment` **must** be a valid pointer to a valid `VkAttachmentReference` structure
- If `preserveAttachmentCount` is not `0`, `pPreserveAttachments` **must** be a valid pointer to an array of `preserveAttachmentCount` `uint32_t` values

Bits which **can** be set in `VkSubpassDescription::flags`, specifying usage of the subpass, are:

```
typedef enum VkSubpassDescriptionFlagBits {
    VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX = 0x00000001,
    VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX = 0x00000002,
    VK_SUBPASS_DESCRIPTION_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkSubpassDescriptionFlagBits;
```

- `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX` specifies that shaders compiled for this subpass write the attributes for all views in a single invocation of each vertex processing stage. All pipelines compiled against a subpass that includes this bit **must** write per-view attributes to the `*PerViewNV[]` shader outputs, in addition to the non-per-view (e.g. `Position`) outputs.
- `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX` specifies that shaders compiled for this subpass use per-view positions which only differ in value in the x component. Per-view viewport mask **can** also be used.

```
typedef VkFlags VkSubpassDescriptionFlags;
```

`VkSubpassDescriptionFlags` is a bitmask type for setting a mask of zero or more

The `VkSubpassDescriptionFlagBits`.

The `VkAttachmentReference` structure is defined as:

```
typedef struct VkAttachmentReference {
    uint32_t attachment;
    VkImageLayout layout;
} VkAttachmentReference;
```

- `attachment` is either an integer value identifying an attachment at the corresponding index in `VkRenderPassCreateInfo::pAttachments`, or `VK_ATTACHMENT_UNUSED` to signify that this attachment is not used.
- `layout` is a `VkImageLayout` value specifying the layout the attachment uses during the subpass.

## Valid Usage

- If `attachment` is not `VK_ATTACHMENT_UNUSED`, `layout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`

## Valid Usage (Implicit)

- `layout` **must** be a valid `VkImageLayout` value

The `VkSubpassDependency` structure is defined as:

```
typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;
```

- `srcSubpass` is the subpass index of the first subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- `dstSubpass` is the subpass index of the second subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `source stage mask`.
- `dstStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `destination stage mask`
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a `source access mask`.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a `destination access mask`.

- `dependencyFlags` is a bitmask of `VkDependencyFlagBits`.

If `srcSubpass` is equal to `dstSubpass` then the `VkSubpassDependency` describes a `subpass self-dependency`, and only constrains the pipeline barriers allowed within a subpass instance. Otherwise, when a render pass instance which includes a subpass dependency is submitted to a queue, it defines a memory dependency between the subpasses identified by `srcSubpass` and `dstSubpass`.

If `srcSubpass` is equal to `VK_SUBPASS_EXTERNAL`, the first `synchronization scope` includes commands that occur earlier in `submission order` than the `vkCmdBeginRenderPass` used to begin the render pass instance. Otherwise, the first set of commands includes all commands submitted as part of the subpass instance identified by `srcSubpass` and any load, store or multisample resolve operations on attachments used in `srcSubpass`. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the `source stage mask` specified by `srcStageMask`.

If `dstSubpass` is equal to `VK_SUBPASS_EXTERNAL`, the second `synchronization scope` includes commands that occur later in `submission order` than the `vkCmdEndRenderPass` used to end the render pass instance. Otherwise, the second set of commands includes all commands submitted as part of the subpass instance identified by `dstSubpass` and any load, store or multisample resolve operations on attachments used in `dstSubpass`. In either case, the second synchronization scope is limited to operations on the pipeline stages determined by the `destination stage mask` specified by `dstStageMask`.

The first `access scope` is limited to access in the pipeline stages determined by the `source stage mask` specified by `srcStageMask`. It is also limited to access types in the `source access mask` specified by `srcAccessMask`.

The second `access scope` is limited to access in the pipeline stages determined by the `destination stage mask` specified by `dstStageMask`. It is also limited to access types in the `destination access mask` specified by `dstAccessMask`.

The `availability and visibility operations` defined by a subpass dependency affect the execution of `image layout transitions` within the render pass.

*Note*

For non-attachment resources, the memory dependency expressed by subpass dependency is nearly identical to that of a [VkMemoryBarrier](#) (with matching `srcAccessMask/dstAccessMask` parameters) submitted as a part of a [vkCmdPipelineBarrier](#) (with matching `srcStageMask/dstStageMask` parameters). The only difference being that its scopes are limited to the identified subpasses rather than potentially affecting everything before and after.



For attachments however, subpass dependencies work more like a [VkImageMemoryBarrier](#) defined similarly to the [VkMemoryBarrier](#) above, the queue family indices set to `VK_QUEUE_FAMILY_IGNORED`, and layouts as follows:

- The equivalent to `oldLayout` is the attachment's layout according to the subpass description for `srcSubpass`.
- The equivalent to `newLayout` is the attachment's layout according to the subpass description for `dstSubpass`.

## Valid Usage

- If the `geometry shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `geometry shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- `srcSubpass` **must** be less than or equal to `dstSubpass`, unless one of them is `VK_SUBPASS_EXTERNAL`, to avoid cyclic dependencies and ensure a valid execution order
- `srcSubpass` and `dstSubpass` **must** not both be equal to `VK_SUBPASS_EXTERNAL`
- If `srcSubpass` is equal to `dstSubpass` and not all of the stages in `srcStageMask` and `dstStageMask` are `framebuffer-space stages`, the `logically latest` pipeline stage in `srcStageMask` **must** be `logically earlier` than or equal to the `logically earliest` pipeline stage in `dstStageMask`
- Any access flag included in `srcAccessMask` **must** be supported by one of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)
- Any access flag included in `dstAccessMask` **must** be supported by one of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)
- If `srcSubpass` equals `dstSubpass`, and `srcStageMask` and `dstStageMask` both include a `framebuffer-space stage`, then `dependencyFlags` **must** include `VK_DEPENDENCY_BY_REGION_BIT`
- If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `srcSubpass` **must** not be equal to `VK_SUBPASS_EXTERNAL`
- If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `dstSubpass` **must** not be equal to `VK_SUBPASS_EXTERNAL`
- If `srcSubpass` equals `dstSubpass` and that subpass has more than one bit set in the view mask, then `dependencyFlags` **must** include `VK_DEPENDENCY_VIEW_LOCAL_BIT`
- If the `mesh shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
- If the `mesh shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

## Valid Usage (Implicit)

- `srcStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `srcStageMask` **must** not be `0`
- `dstStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `dstStageMask` **must** not be `0`
- `srcAccessMask` **must** be a valid combination of `VkAccessFlagBits` values
- `dstAccessMask` **must** be a valid combination of `VkAccessFlagBits` values
- `dependencyFlags` **must** be a valid combination of `VkDependencyFlagBits` values

When multiview is enabled, the execution of the multiple views of one subpass **may** not occur simultaneously or even back-to-back, and rather **may** be interleaved with the execution of other subpasses. The load and store operations apply to attachments on a per-view basis. For example, an attachment using `VK_ATTACHMENT_LOAD_OP_CLEAR` will have each view cleared on first use, but the first use of one view may be temporally distant from the first use of another view.

*Note*

A good mental model for multiview is to think of a multiview subpass as if it were a collection of individual (per-view) subpasses that are logically grouped together and described as a single multiview subpass in the API. Similarly, a multiview attachment can be thought of like several individual attachments that happen to be layers in a single image. A view-local dependency between two multiview subpasses acts like a set of one-to-one dependencies between corresponding pairs of per-view subpasses. A view-global dependency between two multiview subpasses acts like a set of  $N \times M$  dependencies between all pairs of per-view subpasses in the source and destination. Thus, it is a more compact representation which also makes clear the commonality and reuse that is present between views in a subpass. This interpretation motivates the answers to questions like “when does the load op apply” - it is on the first use of each view of an attachment, as if each view were a separate attachment.

If any two subpasses of a render pass activate transform feedback to the same bound transform feedback buffers, a subpass dependency **must** be included (either directly or via some intermediate subpasses) between them.

If there is no subpass dependency from `VK_SUBPASS_EXTERNAL` to the first subpass that uses an attachment, then an implicit subpass dependency exists from `VK_SUBPASS_EXTERNAL` to the first subpass it is used in. The subpass dependency operates as if defined with the following parameters:

```

VkSubpassDependency implicitDependency = {
    .srcSubpass = VK_SUBPASS_EXTERNAL;
    .dstSubpass = firstSubpass; // First subpass attachment is used in
    .srcStageMask = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    .dstStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .srcAccessMask = 0;
    .dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
                    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
                    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dependencyFlags = 0;
};

```

Similarly, if there is no subpass dependency from the last subpass that uses an attachment to `VK_SUBPASS_EXTERNAL`, then an implicit subpass dependency exists from the last subpass it is used in to `VK_SUBPASS_EXTERNAL`. The subpass dependency operates as if defined with the following parameters:

```

VkSubpassDependency implicitDependency = {
    .srcSubpass = lastSubpass; // Last subpass attachment is used in
    .dstSubpass = VK_SUBPASS_EXTERNAL;
    .srcStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .dstStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
    .srcAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
                    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
                    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dstAccessMask = 0;
    .dependencyFlags = 0;
};

```

As subpasses **may** overlap or execute out of order with regards to other subpasses unless a subpass dependency chain describes otherwise, the layout transitions required between subpasses **cannot** be known to an application. Instead, an application provides the layout that each attachment **must** be in at the start and end of a render pass, and the layout it **must** be in during each subpass it is used in. The implementation then **must** execute layout transitions between subpasses in order to guarantee that the images are in the layouts required by each subpass, and in the final layout at the end of the render pass.

Automatic layout transitions apply to the entire image subresource attached to the framebuffer. If the attachment view is a 2D or 2D array view of a 3D image, even if the attachment view only refers to a subset of the slices of the selected mip level of the 3D image, automatic layout transitions apply to the entire subresource referenced which is the entire mip level in this case.

Automatic layout transitions away from the layout used in a subpass happen-after the availability operations for all dependencies with that subpass as the `srcSubpass`.

Automatic layout transitions into the layout used in a subpass happen-before the visibility operations for all dependencies with that subpass as the `dstSubpass`.

Automatic layout transitions away from `initialLayout` happens-after the availability operations for all dependencies with a `srcSubpass` equal to `VK_SUBPASS_EXTERNAL`, where `dstSubpass` uses the attachment that will be transitioned. For attachments created with `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT`, automatic layout transitions away from `initialLayout` happen-after the availability operations for all dependencies with a `srcSubpass` equal to `VK_SUBPASS_EXTERNAL`, where `dstSubpass` uses any aliased attachment.

Automatic layout transitions into `finalLayout` happens-before the visibility operations for all dependencies with a `dstSubpass` equal to `VK_SUBPASS_EXTERNAL`, where `srcSubpass` uses the attachment that will be transitioned. For attachments created with `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT`, automatic layout transitions into `finalLayout` happen-before the visibility operations for all dependencies with a `dstSubpass` equal to `VK_SUBPASS_EXTERNAL`, where `srcSubpass` uses any aliased attachment.

The image layout of the depth aspect of a depth/stencil attachment referring to an image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` is dependent on the last sample locations used to render to the attachment, thus automatic layout transitions use the sample locations state specified in [VkRenderPassSampleLocationsBeginInfoEXT](#).

Automatic layout transitions of an attachment referring to a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` use the sample locations the image subresource range referenced by the attachment was last rendered with. If the current render pass does not use the attachment as a depth/stencil attachment in any subpass that happens-before, the automatic layout transition uses the sample locations state specified in the `sampleLocationsInfo` member of the element of the `VkRenderPassSampleLocationsBeginInfoEXT::pAttachmentInitialSampleLocations` array for which the `attachmentIndex` member equals the attachment index of the attachment, if one is specified. Otherwise, the automatic layout transition uses the sample locations state specified in the `sampleLocationsInfo` member of the element of the `VkRenderPassSampleLocationsBeginInfoEXT::pPostSubpassSampleLocations` array for which the `subpassIndex` member equals the index of the subpass that last used the attachment as a depth/stencil attachment, if one is specified.

If no sample locations state has been specified for an automatic layout transition performed on an attachment referring to a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` the contents of the depth aspect of the depth/stencil attachment become undefined as if the layout of the attachment was transitioned from the `VK_IMAGE_LAYOUT_UNDEFINED` layout.

If two subpasses use the same attachment, and both subpasses use the attachment in a read-only layout, no subpass dependency needs to be specified between those subpasses. If an implementation treats those layouts separately, it **must** insert an implicit subpass dependency between those subpasses to separate the uses in each layout. The subpass dependency operates as if defined with the following parameters:

```

// Used for input attachments
VkPipelineStageFlags inputAttachmentStages = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
VkAccessFlags inputAttachmentAccess = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT;

// Used for depth/stencil attachments
VkPipelineStageFlags depthStencilAttachmentStages =
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
VkAccessFlags depthStencilAttachmentAccess =
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT;

VkSubpassDependency implicitDependency = {
    .srcSubpass = firstSubpass;
    .dstSubpass = secondSubpass;
    .srcStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .dstStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .srcAccessMask = inputAttachmentAccess | depthStencilAttachmentAccess;
    .dstAccessMask = inputAttachmentAccess | depthStencilAttachmentAccess;
    .dependencyFlags = 0;
};

}

```

If a subpass uses the same attachment as both an input attachment and either a color attachment or a depth/stencil attachment, writes via the color or depth/stencil attachment are not automatically made visible to reads via the input attachment, causing a *feedback loop*, except in any of the following conditions:

- If the color components or depth/stencil components read by the input attachment are mutually exclusive with the components written by the color or depth/stencil attachments, then there is no feedback loop. This requires the graphics pipelines used by the subpass to disable writes to color components that are read as inputs via the `colorWriteMask`, and to disable writes to depth/stencil components that are read as inputs via `depthWriteEnable` or `stencilTestEnable`.
- If the attachment is used as an input attachment and depth/stencil attachment only, and the depth/stencil attachment is not written to.
- If a memory dependency is inserted between when the attachment is written and when it is subsequently read by later fragments. [Pipeline barriers](#) expressing a [subpass self-dependency](#) are the only way to achieve this, and one **must** be inserted every time a fragment will read values at a particular sample (x, y, layer, sample) coordinate, if those values have been written since the most recent pipeline barrier; or the since start of the subpass if there have been no pipeline barriers since the start of the subpass.

An attachment used as both an input attachment and a color attachment **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` or `VK_IMAGE_LAYOUT_GENERAL` layout. An attachment used as an input attachment and depth/stencil attachment **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL` layout. An attachment **must** not be used as both a depth/stencil attachment and a color attachment.

A more extensible version of render pass creation is also defined below.

To create a render pass, call:

```
VkResult vkCreateRenderPass2KHR(  
    VkDevice device,  
    const VkRenderPassCreateInfo2KHR* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkRenderPass* pRenderPass);
```

- `device` is the logical device that creates the render pass.
- `pCreateInfo` is a pointer to a `VkRenderPassCreateInfo2KHR` structure describing the parameters of the render pass.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pRenderPass` is a pointer to a `VkRenderPass` handle in which the resulting render pass object is returned.

This command is functionally identical to `vkCreateRenderPass`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkRenderPassCreateInfo2KHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pRenderPass` **must** be a valid pointer to a `VkRenderPass` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkRenderPassCreateInfo2KHR` structure is defined as:

```

typedef struct VkRenderPassCreateInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkRenderPassCreateFlags flags;
    uint32_t attachmentCount;
    const VkAttachmentDescription2KHR* pAttachments;
    uint32_t subpassCount;
    const VkSubpassDescription2KHR* pSubpasses;
    uint32_t dependencyCount;
    const VkSubpassDependency2KHR* pDependencies;
    uint32_t correlatedViewMaskCount;
    const uint32_t* pCorrelatedViewMasks;
} VkRenderPassCreateInfo2KHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **attachmentCount** is the number of attachments used by this render pass.
- **pAttachments** is a pointer to an array of **attachmentCount** **VkAttachmentDescription2KHR** structures describing the attachments used by the render pass.
- **subpassCount** is the number of subpasses to create.
- **pSubpasses** is a pointer to an array of **subpassCount** **VkSubpassDescription2KHR** structures describing each subpass.
- **dependencyCount** is the number of dependencies between pairs of subpasses.
- **pDependencies** is a pointer to an array of **dependencyCount** **VkSubpassDependency2KHR** structures describing dependencies between pairs of subpasses.
- **correlatedViewMaskCount** is the number of correlation masks.
- **pCorrelatedViewMasks** is a pointer to an array of view masks indicating sets of views that **may** be more efficient to render concurrently.

Parameters defined by this structure with the same name as those in [VkRenderPassCreateInfo](#) have the identical effect to those parameters; the child structures are variants of those used in [VkRenderPassCreateInfo](#) which include **sType** and **pNext** parameters, allowing them to be extended.

If the [VkSubpassDescription2KHR::viewMask](#) member of any element of **pSubpasses** is not zero, *multiview* functionality is considered to be enabled for this render pass.

**correlatedViewMaskCount** and **pCorrelatedViewMasks** have the same effect as [VkRenderPassMultiviewCreateInfo::correlationMaskCount](#) and [VkRenderPassMultiviewCreateInfo::pCorrelationMasks](#), respectively.

## Valid Usage

- If any two subpasses operate on attachments with overlapping ranges of the same `VkDeviceMemory` object, and at least one subpass writes to that area of `VkDeviceMemory`, a subpass dependency **must** be included (either directly or via some intermediate subpasses) between them
- If the `attachment` member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment`, or the attachment indexed by any element of `pPreserveAttachments` in any given element of `pSubpasses` is bound to a range of a `VkDeviceMemory` object that overlaps with any other attachment in any subpass (including the same subpass), the `VkAttachmentDescription2KHR` structures describing them **must** include `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` in `flags`
- If the `attachment` member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment`, or any element of `pPreserveAttachments` in any given element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, it **must** be less than `attachmentCount`
- For any member of `pAttachments` with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`
- For any member of `pAttachments` with a `stencilLoadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`.
- For any element of `pDependencies`, if the `srcSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `srcStageMask` member of that dependency **must** be a pipeline stage supported by the `pipeline` identified by the `pipelineBindPoint` member of the source subpass
- For any element of `pDependencies`, if the `dstSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `dstStageMask` member of that dependency **must** be a pipeline stage supported by the `pipeline` identified by the `pipelineBindPoint` member of the destination subpass
- The set of bits included in any element of `pCorrelatedViewMasks` **must** not overlap with the set of bits included in any other element of `pCorrelatedViewMasks`
- If the `VkSubpassDescription2KHR::viewMask` member of all elements of `pSubpasses` is `0`, `correlatedViewMaskCount` **must** be `0`
- The `VkSubpassDescription2KHR::viewMask` member of all elements of `pSubpasses` **must** either all be `0`, or all not be `0`
- If the `VkSubpassDescription2KHR::viewMask` member of all elements of `pSubpasses` is `0`, the `dependencyFlags` member of any element of `pDependencies` **must** not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`
- For any element of `pDependencies` where its `srcSubpass` member equals its `dstSubpass`

member, if the `viewMask` member of the corresponding element of `pSubpasses` includes more than one bit, its `dependencyFlags` member **must** include `VK_DEPENDENCY_VIEW_LOCAL_BIT`

- The `viewMask` member **must** not have a bit set at an index greater than or equal to `VkPhysicalDeviceLimits::maxFramebufferLayers`
- If the `attachment` member of any element of the `pInputAttachments` member of any element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, the `aspectMask` member of that element of `pInputAttachments` **must** only include aspects that are present in images of the format specified by the element of `pAttachments` specified by `attachment`
- The `srcSubpass` member of each element of `pDependencies` **must** be less than `subpassCount`
- The `dstSubpass` member of each element of `pDependencies` **must** be less than `subpassCount`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2_KHR`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkRenderPassFragmentDensityMapCreateInfoEXT`
- `flags` **must** be `0`
- If `attachmentCount` is not `0`, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkAttachmentDescription2KHR` structures
- `pSubpasses` **must** be a valid pointer to an array of `subpassCount` valid `VkSubpassDescription2KHR` structures
- If `dependencyCount` is not `0`, `pDependencies` **must** be a valid pointer to an array of `dependencyCount` valid `VkSubpassDependency2KHR` structures
- If `correlatedViewMaskCount` is not `0`, `pCorrelatedViewMasks` **must** be a valid pointer to an array of `correlatedViewMaskCount` `uint32_t` values
- `subpassCount` **must** be greater than `0`

The `VkAttachmentDescription2KHR` structure is defined as:

```

typedef struct VkAttachmentDescription2KHR {
    VkStructureType          sType;
    const void*             pNext;
    VkAttachmentDescriptionFlags flags;
    VkFormat                 format;
    VkSampleCountFlagBits    samples;
    VkAttachmentLoadOp       loadOp;
    VkAttachmentStoreOp      storeOp;
    VkAttachmentLoadOp       stencilLoadOp;
    VkAttachmentStoreOp      stencilStoreOp;
    VkImageLayout            initialLayout;
    VkImageLayout            finalLayout;
} VkAttachmentDescription2KHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is a bitmask of [VkAttachmentDescriptionFlagBits](#) specifying additional properties of the attachment.
- **format** is a [VkFormat](#) value specifying the format of the image that will be used for the attachment.
- **samples** is the number of samples of the image as defined in [VkSampleCountFlagBits](#).
- **loadOp** is a [VkAttachmentLoadOp](#) value specifying how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used.
- **storeOp** is a [VkAttachmentStoreOp](#) value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
- **stencilLoadOp** is a [VkAttachmentLoadOp](#) value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.
- **stencilStoreOp** is a [VkAttachmentStoreOp](#) value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.
- **initialLayout** is the layout the attachment image subresource will be in when a render pass instance begins.
- **finalLayout** is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

Parameters defined by this structure with the same name as those in [VkAttachmentDescription](#) have the identical effect to those parameters.

If the **separateDepthStencilLayouts** feature is enabled, and **format** is a depth/stencil format, **initialLayout** and **finalLayout** **can** be set to a layout that only specifies the layout of the depth aspect.

If **format** is a depth/stencil format, and **initialLayout** only specifies the initial layout of the depth aspect of the attachment, the initial layout of the stencil aspect is specified by the **stencilInitialLayout** member of an instance of [VkAttachmentDescriptionStencilLayoutKHR](#) in the

`pNext` chain. Otherwise, `initialLayout` describes the initial layout for all relevant image aspects.

If `format` is a depth/stencil format, and `finalLayout` only specifies the final layout of the depth aspect of the attachment, the final layout of the stencil aspect is specified by the `stencilFinalLayout` member of an instance of `VkAttachmentDescriptionStencilLayoutKHR` in the `pNext` chain. Otherwise, `finalLayout` describes the final layout for all relevant image aspects.

## Valid Usage

- `finalLayout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`
- If the `separateDepthStencilLayouts` feature is not enabled, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If the `separateDepthStencilLayouts` feature is not enabled, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If `format` is a color format, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If `format` is a color format, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If `format` is a depth/stencil format which includes both depth and stencil aspects, and `initialLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`, the `pNext` chain **must** include a valid instance of `VkAttachmentDescriptionStencilLayoutKHR`
- If `format` is a depth/stencil format which includes both depth and stencil aspects, and `finalLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`, the `pNext` chain **must** include a valid instance of `VkAttachmentDescriptionStencilLayoutKHR`
- If `format` is a depth/stencil format which includes only the depth aspect, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If `format` is a depth/stencil format which includes only the depth aspect, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- If `format` is a depth/stencil format which includes only the stencil aspect, `initialLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`
- If `format` is a depth/stencil format which includes only the stencil aspect, `finalLayout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`

## Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2_KHR`
- **flags** **must** be a valid combination of `VkAttachmentDescriptionFlagBits` values
- **format** **must** be a valid `VkFormat` value
- **samples** **must** be a valid `VkSampleCountFlagBits` value
- **loadOp** **must** be a valid `VkAttachmentLoadOp` value
- **storeOp** **must** be a valid `VkAttachmentStoreOp` value
- **stencilLoadOp** **must** be a valid `VkAttachmentLoadOp` value
- **stencilStoreOp** **must** be a valid `VkAttachmentStoreOp` value
- **initialLayout** **must** be a valid `VkImageLayout` value
- **finalLayout** **must** be a valid `VkImageLayout` value

The `VkAttachmentDescriptionStencilLayoutKHR` structure is defined as:

```
typedef struct VkAttachmentDescriptionStencilLayoutKHR {
    VkStructureType    sType;
    void*              pNext;
    VkImageLayout      stencilInitialLayout;
    VkImageLayout      stencilFinalLayout;
} VkAttachmentDescriptionStencilLayoutKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **stencilInitialLayout** is the layout the stencil aspect of the attachment image subresource will be in when a render pass instance begins.
- **stencilFinalLayout** is the layout the stencil aspect of the attachment image subresource will be transitioned to when a render pass instance ends.

## Valid Usage

- `stencilInitialLayout` **must** not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` or
- `stencilFinalLayout` **must** not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` or
- `stencilFinalLayout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or  
`VK_IMAGE_LAYOUT_PREINITIALIZED`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT_KHR`
- `stencilInitialLayout` **must** be a valid `VkImageLayout` value
- `stencilFinalLayout` **must** be a valid `VkImageLayout` value

The `VkSubpassDescription2KHR` structure is defined as:

```
typedef struct VkSubpassDescription2KHR {
    VkStructureType             sType;
    const void*                 pNext;
    VkSubpassDescriptionFlags   flags;
    VkPipelineBindPoint         pipelineBindPoint;
    uint32_t                    viewMask;
    uint32_t                    inputAttachmentCount;
    const VkAttachmentReference2KHR* pInputAttachments;
    uint32_t                    colorAttachmentCount;
    const VkAttachmentReference2KHR* pColorAttachments;
    const VkAttachmentReference2KHR* pResolveAttachments;
    const VkAttachmentReference2KHR* pDepthStencilAttachment;
    uint32_t                    preserveAttachmentCount;
    const uint32_t*              pPreserveAttachments;
} VkSubpassDescription2KHR;
```

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkSubpassDescriptionFlagBits` specifying usage of the subpass.
- `pipelineBindPoint` is a `VkPipelineBindPoint` value specifying the pipeline type supported for this subpass.
- `viewMask` is a bitfield of view indices describing which views rendering is broadcast to in this subpass, when multiview is enabled.
- `inputAttachmentCount` is the number of input attachments.
- `pInputAttachments` is a pointer to an array of `VkAttachmentReference2KHR` structures defining the input attachments for this subpass and their layouts.
- `colorAttachmentCount` is the number of color attachments.
- `pColorAttachments` is a pointer to an array of `VkAttachmentReference2KHR` structures defining the color attachments for this subpass and their layouts.
- `pResolveAttachments` is an optional array of `colorAttachmentCount` `VkAttachmentReference2KHR` structures defining the resolve attachments for this subpass and their layouts.
- `pDepthStencilAttachment` is a pointer to a `VkAttachmentReference2KHR` structure specifying the depth/stencil attachment for this subpass and its layout.
- `preserveAttachmentCount` is the number of preserved attachments.
- `pPreserveAttachments` is a pointer to an array of `preserveAttachmentCount` render pass attachment indices identifying attachments that are not used by this subpass, but whose contents **must** be preserved throughout the subpass.

Parameters defined by this structure with the same name as those in `VkSubpassDescription` have the identical effect to those parameters.

`viewMask` has the same effect for the described subpass as `VkRenderPassMultiviewCreateInfo`  
`::pViewMasks` has on each corresponding subpass.

## Valid Usage

- `pipelineBindPoint` **must** be `VK_PIPELINE_BIND_POINT_GRAPHICS`
- `colorAttachmentCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxColorAttachments`
- If the first use of an attachment in this render pass is as an input attachment, and the attachment is not also used as a color or depth/stencil attachment in the same subpass, then `loadOp` **must** not be `VK_ATTACHMENT_LOAD_OP_CLEAR`
- If `pResolveAttachments` is not `NULL`, for each resolve attachment that does not have the value `VK_ATTACHMENT_UNUSED`, the corresponding color attachment **must** not have the value `VK_ATTACHMENT_UNUSED`
- If `pResolveAttachments` is not `NULL`, for each resolve attachment that is not `VK_ATTACHMENT_UNUSED`, the corresponding color attachment **must** not have a sample count of `VK_SAMPLE_COUNT_1_BIT`
- If `pResolveAttachments` is not `NULL`, each resolve attachment that is not `VK_ATTACHMENT_UNUSED` **must** have a sample count of `VK_SAMPLE_COUNT_1_BIT`
- Any given element of `pResolveAttachments` **must** have the same `VkFormat` as its corresponding color attachment
- All attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` **must** have the same sample count
- If the `VK_AMD_mixed_attachment_samples` extension is enabled, all attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` **must** have a sample count that is smaller than or equal to the sample count of `pDepthStencilAttachment` if it is not `VK_ATTACHMENT_UNUSED`
- If neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, and if `pDepthStencilAttachment` is not `VK_ATTACHMENT_UNUSED` and any attachments in `pColorAttachments` are not `VK_ATTACHMENT_UNUSED`, they **must** have the same sample count
- The `attachment` member of any element of `pPreserveAttachments` **must** not be `VK_ATTACHMENT_UNUSED`
- Any given element of `pPreserveAttachments` **must** not also be an element of any other member of the subpass description
- If any attachment is used by more than one `VkAttachmentReference` member, then each use **must** use the same `layout`
- If `flags` includes `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX`, it **must** also include `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX`.
- If the `attachment` member of any element of `pInputAttachments` is not `VK_ATTACHMENT_UNUSED`, then the `aspectMask` member **must** be a valid combination of `VkImageAspectFlagBits`
- If the `attachment` member of any element of `pInputAttachments` is not `VK_ATTACHMENT_UNUSED`, then the `aspectMask` member **must** not be `0`
- If the `attachment` member of any element of `pInputAttachments` is not `VK_ATTACHMENT_UNUSED`, then the `aspectMask` member **must** not include `VK_IMAGE_ASPECT_METADATA_BIT`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2_KHR`
- `flags` must be a valid combination of `VkSubpassDescriptionFlagBits` values
- `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value
- If `inputAttachmentCount` is not `0`, `pInputAttachments` must be a valid pointer to an array of `inputAttachmentCount` valid `VkAttachmentReference2KHR` structures
- If `colorAttachmentCount` is not `0`, `pColorAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference2KHR` structures
- If `colorAttachmentCount` is not `0`, and `pResolveAttachments` is not `NULL`, `pResolveAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference2KHR` structures
- If `pDepthStencilAttachment` is not `NULL`, `pDepthStencilAttachment` must be a valid pointer to a valid `VkAttachmentReference2KHR` structure
- If `preserveAttachmentCount` is not `0`, `pPreserveAttachments` must be a valid pointer to an array of `preserveAttachmentCount` `uint32_t` values

If the `pNext` list of `VkSubpassDescription2KHR` includes a `VkSubpassDescriptionDepthStencilResolveKHR` structure, then that structure describes multisample resolve operations for the depth/stencil attachment in a subpass.

The `VkSubpassDescriptionDepthStencilResolveKHR` structure is defined as:

```
typedef struct VkSubpassDescriptionDepthStencilResolveKHR {  
    VkStructureType           sType;  
    const void*               pNext;  
    VkResolveModeFlagBitsKHR  depthResolveMode;  
    VkResolveModeFlagBitsKHR  stencilResolveMode;  
    const VkAttachmentReference2KHR* pDepthStencilResolveAttachment;  
} VkSubpassDescriptionDepthStencilResolveKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `depthResolveMode` is a bitmask of `VkResolveModeFlagBitsKHR` describing the depth resolve mode.
- `stencilResolveMode` is a bitmask of `VkResolveModeFlagBitsKHR` describing the stencil resolve mode.
- `pDepthStencilResolveAttachment` is an optional `VkAttachmentReference` structure defining the depth/stencil resolve attachment for this subpass and its layout.

## Valid Usage

- If `pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, `pDepthStencilAttachment` **must** not have the value `VK_ATTACHMENT_UNUSED`
- If `pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, `depthResolveMode` and `stencilResolveMode` **must** not both be `VK_RESOLVE_MODE_NONE_KHR`
- If `pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, `pDepthStencilAttachment` **must** not have a sample count of `VK_SAMPLE_COUNT_1_BIT`
- If `pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED`, `pDepthStencilResolveAttachment` **must** have a sample count of `VK_SAMPLE_COUNT_1_BIT`
- If `pDepthStencilResolveAttachment` is not `NULL` and does not have the value `VK_ATTACHMENT_UNUSED` then it **must** have a format whose features contain `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If the `VkFormat` of `pDepthStencilResolveAttachment` has a depth component, then the `VkFormat` of `pDepthStencilAttachment` **must** have a depth component with the same number of bits and numerical type
- If the `VkFormat` of `pDepthStencilResolveAttachment` has a stencil component, then the `VkFormat` of `pDepthStencilAttachment` **must** have a stencil component with the same number of bits and numerical type
- The value of `depthResolveMode` **must** be one of the bits set in `VkPhysicalDeviceDepthStencilResolvePropertiesKHR::supportedDepthResolveModes` or `VK_RESOLVE_MODE_NONE_KHR`
- The value of `stencilResolveMode` **must** be one of the bits set in `VkPhysicalDeviceDepthStencilResolvePropertiesKHR::supportedStencilResolveModes` or `VK_RESOLVE_MODE_NONE_KHR`
- If the `VkFormat` of `pDepthStencilResolveAttachment` has both depth and stencil components, `VkPhysicalDeviceDepthStencilResolvePropertiesKHR::independentResolve` is `VK_FALSE`, and `VkPhysicalDeviceDepthStencilResolvePropertiesKHR::independentResolveNone` is `VK_FALSE`, then the values of `depthResolveMode` and `stencilResolveMode` **must** be identical
- If the `VkFormat` of `pDepthStencilResolveAttachment` has both depth and stencil components, `VkPhysicalDeviceDepthStencilResolvePropertiesKHR::independentResolve` is `VK_FALSE`, and `VkPhysicalDeviceDepthStencilResolvePropertiesKHR::independentResolveNone` is `VK_TRUE`, then the values of `depthResolveMode` and `stencilResolveMode` **must** be identical or one of them **must** be `VK_RESOLVE_MODE_NONE_KHR`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE_KHR`
- `depthResolveMode` **must** be a valid `VkResolveModeFlagBitsKHR` value
- `stencilResolveMode` **must** be a valid `VkResolveModeFlagBitsKHR` value
- If `pDepthStencilResolveAttachment` is not `NULL`, `pDepthStencilResolveAttachment` **must** be a valid pointer to a valid `VkAttachmentReference2KHR` structure

Possible values of `VkSubpassDescriptionDepthStencilResolveKHR::depthResolveMode` and `stencilResolveMode`, specifying the depth and stencil resolve modes, are:

```
typedef enum VkResolveModeFlagBitsKHR {
    VK_RESOLVE_MODE_NONE_KHR = 0,
    VK_RESOLVE_MODE_SAMPLE_ZERO_BIT_KHR = 0x00000001,
    VK_RESOLVE_MODE_AVERAGE_BIT_KHR = 0x00000002,
    VK_RESOLVE_MODE_MIN_BIT_KHR = 0x00000004,
    VK_RESOLVE_MODE_MAX_BIT_KHR = 0x00000008,
    VK_RESOLVE_MODE_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkResolveModeFlagBitsKHR;
```

- `VK_RESOLVE_MODE_NONE_KHR` indicates that no resolve operation is done.
- `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT_KHR` indicates that result of the resolve operation is equal to the value of sample 0.
- `VK_RESOLVE_MODE_AVERAGE_BIT_KHR` indicates that result of the resolve operation is the average of the sample values.
- `VK_RESOLVE_MODE_MIN_BIT_KHR` indicates that result of the resolve operation is the minimum of the sample values.
- `VK_RESOLVE_MODE_MAX_BIT_KHR` indicates that result of the resolve operation is the maximum of the sample values.

```
typedef VkFlags VkResolveModeFlagsKHR;
```

`VkResolveModeFlagsKHR` is a bitmask type for setting a mask of zero or more `VkResolveModeFlagBitsKHR`.

The `VkAttachmentReference2KHR` structure is defined as:

```
typedef struct VkAttachmentReference2KHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t attachment;
    VkImageLayout layout;
    VkImageAspectFlags aspectMask;
} VkAttachmentReference2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `attachment` is either an integer value identifying an attachment at the corresponding index in `VkRenderPassCreateInfo::pAttachments`, or `VK_ATTACHMENT_UNUSED` to signify that this attachment is not used.
- `layout` is a `VkImageLayout` value specifying the layout the attachment uses during the subpass.
- `aspectMask` is a mask of which aspect(s) **can** be accessed within the specified subpass as an input attachment.

Parameters defined by this structure with the same name as those in `VkAttachmentReference` have the identical effect to those parameters.

`aspectMask` has the same effect for the described attachment as `VkInputAttachmentAspectReference::aspectMask` has on each corresponding attachment. It is ignored when this structure is used to describe anything other than an input attachment reference.

If the `separateDepthStencilLayouts` feature is enabled, and `attachment` has a depth/stencil format, `layout` **can** be set to a layout that only specifies the layout of the depth aspect.

If `layout` only specifies the layout of the depth aspect of the attachment, the layout of the stencil aspect is specified by the `stencilLayout` member of an instance of `VkAttachmentReferenceStencilLayoutKHR` in the `pNext` chain. Otherwise, `layout` describes the layout for all relevant image aspects.

## Valid Usage

- If `attachment` is not `VK_ATTACHMENT_UNUSED`, `layout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`
- If the `separateDepthStencilLayouts` feature is not enabled, and `attachment` is not `VK_ATTACHMENT_UNUSED`, `layout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`, or
- If `attachment` is not `VK_ATTACHMENT_UNUSED`, and `aspectMask` includes `VK_IMAGE_ASPECT_COLOR_BIT`, `layout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`, or
- If `attachment` is not `VK_ATTACHMENT_UNUSED`, and `aspectMask` includes both `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT`, and `layout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` or  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`, the `pNext` chain **must** include a valid instance of `VkAttachmentReferenceStencilLayoutKHR`
- If `attachment` is not `VK_ATTACHMENT_UNUSED`, and `aspectMask` includes only `VK_IMAGE_ASPECT_DEPTH_BIT` then `layout` **must** not be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` or
- If `attachment` is not `VK_ATTACHMENT_UNUSED`, and `aspectMask` includes only `VK_IMAGE_ASPECT_STENCIL_BIT` then `layout` **must** not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`,  
`VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR` or

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2_KHR`
- `layout` **must** be a valid `VkImageLayout` value

The `VkAttachmentReferenceStencilLayoutKHR` structure is defined as:

```
typedef struct VkAttachmentReferenceStencilLayoutKHR {
    VkStructureType sType;
    void* pNext;
    VkImageLayout stencilLayout;
} VkAttachmentReferenceStencilLayoutKHR;
```

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `stencilLayout` is a `VkImageLayout` value specifying the layout the stencil aspect of the attachment uses during the subpass.

## Valid Usage

- `stencilLayout` **must** not be `VK_IMAGE_LAYOUT_UNDEFINED`, `VK_IMAGE_LAYOUT_PREINITIALIZED`, `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`
- or

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT_KHR`
- `stencilLayout` **must** be a valid `VkImageLayout` value

The `VkSubpassDependency2KHR` structure is defined as:

```
typedef struct VkSubpassDependency2KHR {
    VkStructureType          sType;
    const void*              pNext;
    uint32_t                 srcSubpass;
    uint32_t                 dstSubpass;
    VkPipelineStageFlags     srcStageMask;
    VkPipelineStageFlags     dstStageMask;
    VkAccessFlags            srcAccessMask;
    VkAccessFlags            dstAccessMask;
    VkDependencyFlags        dependencyFlags;
    int32_t                  viewOffset;
} VkSubpassDependency2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcSubpass` is the subpass index of the first subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- `dstSubpass` is the subpass index of the second subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `source stage mask`.
- `dstStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the `destination stage mask`
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a `source access mask`.

- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a [destination access mask](#).
- `dependencyFlags` is a bitmask of `VkDependencyFlagBits`.
- `viewOffset` controls which views in the source subpass the views in the destination subpass depend on.

Parameters defined by this structure with the same name as those in `VkSubpassDependency` have the identical effect to those parameters.

`viewOffset` has the same effect for the described subpass dependency as `VkRenderPassMultiviewCreateInfo::pViewOffsets` has on each corresponding subpass dependency.

## Valid Usage

- If the `geometry shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `geometry shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If the `tessellation shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- `srcSubpass` **must** be less than or equal to `dstSubpass`, unless one of them is `VK_SUBPASS_EXTERNAL`, to avoid cyclic dependencies and ensure a valid execution order
- `srcSubpass` and `dstSubpass` **must** not both be equal to `VK_SUBPASS_EXTERNAL`
- If `srcSubpass` is equal to `dstSubpass` and not all of the stages in `srcStageMask` and `dstStageMask` are `framebuffer-space stages`, the `logically latest` pipeline stage in `srcStageMask` **must** be `logically earlier` than or equal to the `logically earliest` pipeline stage in `dstStageMask`
- Any access flag included in `srcAccessMask` **must** be supported by one of the pipeline stages in `srcStageMask`, as specified in the [table of supported access types](#)
- Any access flag included in `dstAccessMask` **must** be supported by one of the pipeline stages in `dstStageMask`, as specified in the [table of supported access types](#)
- If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `srcSubpass` **must** not be equal to `VK_SUBPASS_EXTERNAL`
- If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `dstSubpass` **must** not be equal to `VK_SUBPASS_EXTERNAL`
- If `srcSubpass` equals `dstSubpass`, and `srcStageMask` and `dstStageMask` both include a `framebuffer-space stage`, then `dependencyFlags` **must** include `VK_DEPENDENCY_BY_REGION_BIT`
- If `viewOffset` is not equal to `0`, `srcSubpass` **must** not be equal to `dstSubpass`
- If `dependencyFlags` does not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `viewOffset` **must** be `0`
- If `viewOffset` is not `0`, `srcSubpass` **must** not be equal to `dstSubpass`.
- If the `mesh shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
- If the `mesh shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`
- If the `task shaders` feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2_KHR`
- `srcStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `srcStageMask` **must** not be `0`
- `dstStageMask` **must** be a valid combination of `VkPipelineStageFlagBits` values
- `dstStageMask` **must** not be `0`
- `srcAccessMask` **must** be a valid combination of `VkAccessFlagBits` values
- `dstAccessMask` **must** be a valid combination of `VkAccessFlagBits` values
- `dependencyFlags` **must** be a valid combination of `VkDependencyFlagBits` values

To destroy a render pass, call:

```
void vkDestroyRenderPass(  
    VkDevice                                     device,  
    VkRenderPass                                renderPass,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the render pass.
- `renderPass` is the handle of the render pass to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `renderPass` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `renderPass` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `renderPass` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `renderPass` is not `VK_NULL_HANDLE`, `renderPass` **must** be a valid `VkRenderPass` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `renderPass` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `renderPass` must be externally synchronized

## 7.2. Render Pass Compatibility

Framebuffers and graphics pipelines are created based on a specific render pass object. They **must** only be used with that render pass object, or one compatible with it.

Two attachment references are compatible if they have matching format and sample count, or are both `VK_ATTACHMENT_UNUSED` or the pointer that would contain the reference is `NULL`.

Two arrays of attachment references are compatible if all corresponding pairs of attachments are compatible. If the arrays are of different lengths, attachment references not present in the smaller array are treated as `VK_ATTACHMENT_UNUSED`.

Two render passes are compatible if their corresponding color, input, resolve, and depth/stencil attachment references are compatible and if they are otherwise identical except for:

- Initial and final image layout in attachment descriptions
- Load and store operations in attachment descriptions
- Image layout in attachment references

As an additional special case, if two render passes have a single subpass, the resolve attachment reference and depth/stencil resolve mode compatibility requirements are ignored.

A framebuffer is compatible with a render pass if it was created using the same render pass or a compatible render pass.

## 7.3. Framebuffers

Render passes operate in conjunction with *framebuffers*. Framebuffers represent a collection of specific memory attachments that a render pass instance uses.

Framebuffers are represented by `VkFramebuffer` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFramebuffer)
```

To create a framebuffer, call:

```
VkResult vkCreateFramebuffer(  
    VkDevice                                     device,  
    const VkFramebufferCreateInfo*                pCreateInfo,  
    const VkAllocationCallbacks*                 pAllocator,  
    VkFramebuffer*                            pFramebuffer);
```

- `device` is the logical device that creates the framebuffer.
- `pCreateInfo` is a pointer to a `VkFramebufferCreateInfo` structure describing additional information about framebuffer creation.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pFramebuffer` is a pointer to a `VkFramebuffer` handle in which the resulting framebuffer object is returned.

## Valid Usage

- If `pCreateInfo->flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, and `attachmentCount` is not `0`, each element of `pCreateInfo->pAttachments` **must** have been created on `device`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkFramebufferCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pFramebuffer` **must** be a valid pointer to a `VkFramebuffer` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkFramebufferCreateInfo` structure is defined as:

```

typedef struct VkFramebufferCreateInfo {
    VkStructureType          sType;
    const void*             pNext;
    VkFramebufferCreateFlags flags;
    VkRenderPass              renderPass;
    uint32_t                  attachmentCount;
    const VkImageView*       pAttachments;
    uint32_t                  width;
    uint32_t                  height;
    uint32_t                  layers;
} VkFramebufferCreateInfo;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is a bitmask of [VkFramebufferCreateFlagBits](#)
- **renderPass** is a render pass defining what render passes the framebuffer will be compatible with. See [Render Pass Compatibility](#) for details.
- **attachmentCount** is the number of attachments.
- **pAttachments** is a pointer to an array of [VkImageView](#) handles, each of which will be used as the corresponding attachment in a render pass instance. If **flags** includes **VK\_FRAMEBUFFER\_CREATE\_IMAGELESS\_BIT\_KHR**, this parameter is ignored.
- **width**, **height** and **layers** define the dimensions of the framebuffer. If the render pass uses multiview, then **layers** **must** be one and each attachment requires a number of layers that is greater than the maximum bit index set in the view mask in the subpasses in which it is used.

Applications **must** ensure that all accesses to memory that backs image subresources used as attachments in a given renderpass instance either happen-before the [load operations](#) for those attachments, or happen-after the [store operations](#) for those attachments.

For depth/stencil attachments, each aspect **can** be used separately as attachments and non-attachments as long as the non-attachment accesses are also via an image subresource in either the **VK\_IMAGE\_LAYOUT\_DEPTH\_READ\_ONLY\_STENCIL\_ATTACHMENT\_OPTIMAL** layout or the **VK\_IMAGE\_LAYOUT\_DEPTH\_ATTACHMENT\_STENCIL\_READ\_ONLY\_OPTIMAL** layout, and the attachment resource uses whichever of those two layouts the image accesses do not. Use of non-attachment aspects in this case is only well defined if the attachment is used in the subpass where the non-attachment access is being made, or the layout of the image subresource is constant throughout the entire render pass instance, including the **initialLayout** and **finalLayout**.

#### Note



These restrictions mean that the render pass has full knowledge of all uses of all of the attachments, so that the implementation is able to make correct decisions about when and how to perform layout transitions, when to overlap execution of subpasses, etc.

It is legal for a subpass to use no color or depth/stencil attachments, and rather use shader side

effects such as image stores and atomics to produce an output. In this case, the subpass continues to use the `width`, `height`, and `layers` of the framebuffer to define the dimensions of the rendering area, and the `rasterizationSamples` from each pipeline's `VkPipelineMultisampleStateCreateInfo` to define the number of samples used in rasterization; however, if `VkPhysicalDeviceFeatures::variableMultisampleRate` is `VK_FALSE`, then all pipelines to be bound with a given zero-attachment subpass **must** have the same value for `VkPipelineMultisampleStateCreateInfo::rasterizationSamples`.

## Valid Usage

- `attachmentCount` **must** be equal to the attachment count specified in `renderPass`
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, and `attachmentCount` is not `0`, `pAttachments` must be a valid pointer to an array of `attachmentCount` valid `VkImageView` handles
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` that is used as a color attachment or resolve attachment by `renderPass` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` that is used as a depth/stencil attachment by `renderPass` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` that is used as a depth/stencil resolve attachment by `renderPass` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` that is used as an input attachment by `renderPass` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`
- Each element of `pAttachments` that is used as a fragment density map attachment by `renderPass` **must** not have been created with a `flags` value including `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.
- If `renderPass` has a fragment density map attachment and `non-subsample image feature` is not enabled, each element of `pAttachments` **must** have been created with a `flags` value including `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT` unless that element is the fragment density map attachment.
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` **must** have been created with a `VkFormat` value that matches the `VkFormat` specified by the corresponding `VkAttachmentDescription` in `renderPass`
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` **must** have been created with a `samples` value that matches the `samples` value specified by the corresponding `VkAttachmentDescription` in `renderPass`
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` **must** have dimensions at least as large as the corresponding framebuffer dimension except for any element that is referenced by `fragmentDensityMapAttachment`
- If `renderPass` was specified with non-zero view masks, each element of `pAttachments` that is not referenced by `fragmentDensityMapAttachment` **must** have a `layerCount` greater than the index of the most significant bit set in any of those view masks
- If `renderPass` was specified with non-zero view masks, each element of `pAttachments` that is referenced by `fragmentDensityMapAttachment` **must** have a `layerCount` equal to `1` or greater than the index of the most significant bit set in any of those view masks
- If `renderPass` was not specified with non-zero view masks, each element of `pAttachments` that is referenced by `fragmentDensityMapAttachment` **must** have a `layerCount` equal to `1`

- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, an element of `pAttachments` that is referenced by `fragmentDensityMapAttachment` **must** have a width at least as large as  $\lceil \frac{\text{width}}{\maxFragmentDensityTexelSize_{\text{width}}} \rceil$
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, an element of `pAttachments` that is referenced by `fragmentDensityMapAttachment` **must** have a height at least as large as  $\lceil \frac{\text{height}}{\maxFragmentDensityTexelSize_{\text{height}}} \rceil$
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` **must** only specify a single mip level
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` **must** have been created with the identity swizzle
- `width` **must** be greater than `0`.
- `width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferWidth`
- `height` **must** be greater than `0`.
- `height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferHeight`
- `layers` **must** be greater than `0`.
- `layers` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferLayers`
- If `renderPass` was specified with non-zero view masks, `layers` **must** be `1`
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of `pAttachments` that is a 2D or 2D array image view taken from a 3D image **must** not be a depth/stencil format
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, and `attachmentCount` is not `0`, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkImageView` handles
- If the `imageless framebuffer` feature is not enabled, `flags` **must** not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `pNext` chain **must** include an instance of `VkFramebufferAttachmentsCreateInfoKHR`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `attachmentImageInfoCount` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain **must** be equal to either zero or `attachmentCount`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `width` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain **must** be greater than or equal to `width`, except for any element that is referenced by `VkRenderPassFragmentDensityMapCreateInfoEXT::fragmentDensityMapAttachment` in `renderPass`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `height` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain **must** be greater than or equal to `height`, except for any element that is referenced by `VkRenderPassFragmentDensityMapCreateInfoEXT::fragmentDensityMapAttachment` in `renderPass`

- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `width` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain that is referenced by `VkRenderPassFragmentDensityMapCreateInfoEXT::fragmentDensityMapAttachment` in `renderPass` **must** be greater than or equal to  $\lceil \frac{\text{width}}{\maxFragmentDensityTexelSize_{\text{width}}} \rceil$
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `height` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain that is referenced by `VkRenderPassFragmentDensityMapCreateInfoEXT::fragmentDensityMapAttachment` in `renderPass` **must** be greater than or equal to  $\lceil \frac{\text{height}}{\maxFragmentDensityTexelSize_{\text{height}}} \rceil$
- If multiview is enabled for `renderPass`, and `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `layerCount` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain **must** be greater than the maximum bit index set in the view mask in the subpasses in which it is used in `renderPass`
- If multiview is not enabled for `renderPass`, and `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `layerCount` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain **must** be greater than or equal to `layers`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `usage` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain that refers to an attachment used as a color attachment or resolve attachment by `renderPass` **must** include `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `usage` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain that refers to an attachment used as a depth/stencil attachment by `renderPass` **must** include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `usage` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain that refers to an attachment used as a depth/stencil resolve attachment by `renderPass` **must** include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `usage` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain that refers to an attachment used as an input attachment by `renderPass` **must** include `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`
- If `flags` includes `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, at least one element of the `pViewFormats` member of any element of the `pAttachmentImageInfos` member of an instance of `VkFramebufferAttachmentsCreateInfoKHR` in the `pNext` chain **must** be equal to the corresponding value of `VkAttachmentDescription::format` used to create `renderPass`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkFramebufferAttachmentsCreateInfoKHR`
- `flags` **must** be a valid combination of `VkFramebufferCreateFlagBits` values
- `renderPass` **must** be a valid `VkRenderPass` handle
- Both of `renderPass`, and the elements of `pAttachments` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

The `VkFramebufferAttachmentsCreateInfoKHR` structure is defined as:

```
typedef struct VkFramebufferAttachmentsCreateInfoKHR {  
    VkStructureType sType;  
    const void* pNext;  
    uint32_t attachmentImageInfoCount;  
    const VkFramebufferAttachmentImageInfoKHR* pAttachmentImageInfos;  
} VkFramebufferAttachmentsCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `attachmentImageInfoCount` is the number of attachments being described.
- `pAttachmentImageInfos` is a pointer to an array of `VkFramebufferAttachmentImageInfoKHR` instances, each of which describes a number of parameters of the corresponding attachment in a render pass instance.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO_KHR`
- If `attachmentImageInfoCount` is not `0`, `pAttachmentImageInfos` **must** be a valid pointer to an array of `attachmentImageInfoCount` valid `VkFramebufferAttachmentImageInfoKHR` structures

The `VkFramebufferAttachmentImageInfoKHR` structure is defined as:

```

typedef struct VkFramebufferAttachmentImageInfoKHR {
    VkStructureType      sType;
    const void*        pNext;
    VkImageCreateFlags   flags;
    VkImageUsageFlags    usage;
    uint32_t             width;
    uint32_t             height;
    uint32_t             layerCount;
    uint32_t             viewFormatCount;
    const VkFormat*     pViewFormats;
} VkFramebufferAttachmentImageInfoKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is a bitmask of [VkImageCreateFlagBits](#), matching the value of [VkImageCreateInfo::flags](#) used to create an image that will be used with this framebuffer.
- **usage** is a bitmask of [VkImageUsageFlagBits](#), matching the value of [VkImageCreateInfo::usage](#) used to create an image used with this framebuffer.
- **width** is the width of the image view used for rendering.
- **height** is the height of the image view used for rendering.
- **viewFormatCount** is the number of entries in the **pViewFormats** array, matching the value of [VkImageFormatListCreateInfoKHR::viewFormatCount](#) used to create an image used with this framebuffer.
- **pViewFormats** is an array which lists of all formats which **can** be used when creating views of the image, matching the value of [VkImageFormatListCreateInfoKHR::pViewFormats](#) used to create an image used with this framebuffer.

Images that **can** be used with the framebuffer when beginning a render pass, as specified by [VkRenderPassAttachmentBeginInfoKHR](#), **must** be created with parameters that are identical to those specified here.

## Valid Usage (Implicit)

- **sType must** be [VK\\_STRUCTURE\\_TYPE\\_FRAMEBUFFER\\_ATTACHMENT\\_IMAGE\\_INFO\\_KHR](#)
- **pNext must** be **NULL**
- **flags must** be a valid combination of [VkImageCreateFlagBits](#) values
- **usage must** be a valid combination of [VkImageUsageFlagBits](#) values
- **usage must** not be **0**
- If **viewFormatCount** is not **0**, **pViewFormats must** be a valid pointer to an array of **viewFormatCount** valid [VkFormat](#) values

Bits which **can** be set in [VkFramebufferCreateInfo::flags](#) to specify options for framebuffers are:

```
typedef enum VkFramebufferCreateFlagBits {
    VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR = 0x00000001,
    VK_FRAMEBUFFER_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkFramebufferCreateFlagBits;
```

- `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR` specifies that image views are not specified, and only attachment compatibility information will be provided via an instance of `VkFramebufferAttachmentImageInfoKHR`.

```
typedef VkFlags VkFramebufferCreateFlags;
```

`VkFramebufferCreateFlags` is a bitmask type for setting a mask of zero or more `VkFramebufferCreateFlagBits`.

To destroy a framebuffer, call:

```
void vkDestroyFramebuffer(
    VkDevice                               device,
    VkFramebuffer                          framebuffer,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the framebuffer.
- `framebuffer` is the handle of the framebuffer to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `framebuffer` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `framebuffer` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `framebuffer` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `framebuffer` is not `VK_NULL_HANDLE`, `framebuffer` **must** be a valid `VkFramebuffer` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `framebuffer` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `framebuffer` must be externally synchronized

## 7.4. Render Pass Commands

An application records the commands for a render pass instance one subpass at a time, by beginning a render pass instance, iterating over the subpasses to record commands for that subpass, and then ending the render pass instance.

To begin a render pass instance, call:

```
void vkCmdBeginRenderPass(  
    VkCommandBuffer  
    const VkRenderPassBeginInfo*  
    VkSubpassContents  
        commandBuffer,  
        pRenderPassBegin,  
        contents);
```

- `commandBuffer` is the command buffer in which to record the command.
- `pRenderPassBegin` is a pointer to a `VkRenderPassBeginInfo` structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.
- `contents` is a `VkSubpassContents` value specifying how the commands in the first subpass will be provided.

After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.

## Valid Usage

- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, or  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_SRC_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_DST_BIT`
- If any of the `initialLayout` members of the `VkAttachmentDescription` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is not `VK_IMAGE_LAYOUT_UNDEFINED`, then each such `initialLayout` **must** be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin`
- The `srcStageMask` and `dstStageMask` members of any element of the `pDependencies` member of `VkRenderPassCreateInfo` used to create `renderPass` **must** be supported by the capabilities of the queue family identified by the `queueFamilyIndex` member of the

[VkCommandPoolCreateInfo](#) used to create the command pool which `commandBuffer` was allocated from

- For any attachment in `framebuffer` that is used by `renderPass` and is bound to memory locations that are also bound to another attachment used by `renderPass`, and if at least one of those uses causes either attachment to be written to, both attachments **must** have had the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` set

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid [VkCommandBuffer](#) handle
- `pRenderPassBegin` **must** be a valid pointer to a valid [VkRenderPassBeginInfo](#) structure
- `contents` **must** be a valid [VkSubpassContents](#) value
- `commandBuffer` **must** be in the [recording state](#)
- The [VkCommandPool](#) that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called outside of a render pass instance
- `commandBuffer` **must** be a primary [VkCommandBuffer](#)

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the [VkCommandPool](#) that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Outside	Graphics	Graphics

Alternatively to begin a render pass, call:

```
void vkCmdBeginRenderPass2KHR(  
    VkCommandBuffer  
    const VkRenderPassBeginInfo*  
    const VkSubpassBeginInfoKHR*  
                                commandBuffer,  
                                pRenderPassBegin,  
                                pSubpassBeginInfo);
```

- `commandBuffer` is the command buffer in which to record the command.
- `pRenderPassBegin` is a pointer to a [VkRenderPassBeginInfo](#) structure specifying the render pass

to begin an instance of, and the framebuffer the instance uses.

- `pSubpassBeginInfo` is a pointer to a `VkSubpassBeginInfoKHR` structure containing information about the subpass which is about to begin rendering.

After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.

## Valid Usage

- Both the `framebuffer` and `renderPass` members of `pRenderPassBegin` **must** have been created on the same `VkDevice` that `commandBuffer` was allocated on
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`,  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, or  
`VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_SRC_BIT`
- If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_DST_BIT`
- If any of the `initialLayout` members of the `VkAttachmentDescription` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is not `VK_IMAGE_LAYOUT_UNDEFINED`, then each such `initialLayout` **must** be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin`
- The `srcStageMask` and `dstStageMask` members of any element of the `pDependencies` member

of `VkRenderPassCreateInfo` used to create `renderPass` **must** be supported by the capabilities of the queue family identified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` used to create the command pool which `commandBuffer` was allocated from

- For any attachment in `framebuffer` that is used by `renderPass` and is bound to memory locations that are also bound to another attachment used by `renderPass`, and if at least one of those uses causes either attachment to be written to, both attachments **must** have had the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` set

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pRenderPassBegin` **must** be a valid pointer to a valid `VkRenderPassBeginInfo` structure
- `pSubpassBeginInfo` **must** be a valid pointer to a valid `VkSubpassBeginInfoKHR` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called outside of a render pass instance
- `commandBuffer` **must** be a primary `VkCommandBuffer`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Outside	Graphics	Graphics

The `VkRenderPassBeginInfo` structure is defined as:

```

typedef struct VkRenderPassBeginInfo {
    VkStructureType      sType;
    const void*        pNext;
    VkRenderPass         renderPass;
    VkFramebuffer       framebuffer;
    VkRect2D             renderArea;
    uint32_t              clearValueCount;
    const VkClearValue* pClearValues;
} VkRenderPassBeginInfo;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **renderPass** is the render pass to begin an instance of.
- **framebuffer** is the framebuffer containing the attachments that are used with the render pass.
- **renderArea** is the render area that is affected by the render pass instance, and is described in more detail below.
- **clearValueCount** is the number of elements in **pClearValues**.
- **pClearValues** is a pointer to an array of **clearValueCount** **VkClearValue** structures that contains clear values for each attachment, if the attachment uses a **loadOp** value of **VK\_ATTACHMENT\_LOAD\_OP\_CLEAR** or if the attachment has a depth/stencil format and uses a **stencilLoadOp** value of **VK\_ATTACHMENT\_LOAD\_OP\_CLEAR**. The array is indexed by attachment number. Only elements corresponding to cleared attachments are used. Other elements of **pClearValues** are ignored.

**renderArea** is the render area that is affected by the render pass instance. The effects of attachment load, store and multisample resolve operations are restricted to the pixels whose x and y coordinates fall within the render area on all attachments. The render area extends to all layers of **framebuffer**. The application **must** ensure (using scissor if necessary) that all rendering is contained within the render area. The render area **must** be contained within the framebuffer dimensions.

When multiview is enabled, the resolve operation at the end of a subpass applies to all views in the view mask.

*Note*



There **may** be a performance cost for using a render area smaller than the framebuffer, unless it matches the render area granularity for the render pass.

## Valid Usage

- `clearValueCount` **must** be greater than the largest attachment index in `renderPass` that specifies a `loadOp` (or `stencilLoadOp`, if the attachment has a depth/stencil format) of `VK_ATTACHMENT_LOAD_OP_CLEAR`
- `renderPass` **must** be `compatible` with the `renderPass` member of the `VkFramebufferCreateInfo` structure specified when creating `framebuffer`.
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that did not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, and the `pNext` chain includes an instance of `VkRenderPassAttachmentBeginInfoKHR`, its `attachmentCount` **must** be zero
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, the `attachmentCount` of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be equal to the value of `VkFramebufferAttachmentsCreateInfoKHR::attachmentImageInfoCount` used to create `framebuffer`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** have been created on the same `VkDevice` as `framebuffer` and `renderPass`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` of an image created with a value of `VkImageCreateInfo::flags` equal to the `flags` member of the corresponding element of `VkFramebufferAttachmentsCreateInfoKHR::pAttachments` used to create `framebuffer`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` of an image created with a value of `VkImageCreateInfo::usage` equal to the `usage` member of the corresponding element of `VkFramebufferAttachmentsCreateInfoKHR::pAttachments` used to create `framebuffer`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` with a width equal to the `width` member of the corresponding element of `VkFramebufferAttachmentsCreateInfoKHR::pAttachments` used to create `framebuffer`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` with a height equal to the `height` member of the corresponding element of `VkFramebufferAttachmentsCreateInfoKHR::pAttachments` used to create `framebuffer`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a

`VkImageView` of an image created with a value of `VkImageViewCreateInfo::subresourceRange.pName`:`layerCount` equal to the `layerCount` member of the corresponding element of `VkFramebufferAttachmentsCreateInfoKHR::pAttachments` used to create `framebuffer`

- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` of an image created with a value of `VkImageFormatListCreateInfoKHR::viewFormatCount` equal to the `viewFormatCount` member of the corresponding element of `VkFramebufferAttachmentsCreateInfoKHR::pAttachments` used to create `framebuffer`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` of an image created with a set of elements in `VkImageFormatListCreateInfoKHR::pViewFormats` equal to the set of elements in the `pViewFormats` member of the corresponding element of `VkFramebufferAttachmentsCreateInfoKHR::pAttachments` used to create `framebuffer`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` of an image created with a value of `VkImageViewCreateInfo::format` equal to the corresponding value of `VkAttachmentDescription::format` in `renderPass`
- If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`, each element of the `pAttachments` member of an instance of `VkRenderPassAttachmentBeginInfoKHR` included in the `pNext` chain **must** be a `VkImageView` of an image created with a value of `VkImageCreateInfo::samples` equal to the corresponding value of `VkAttachmentDescription::samples` in `renderPass`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkDeviceGroupRenderPassBeginInfo`, `VkRenderPassAttachmentBeginInfoKHR`, or `VkRenderPassSampleLocationsBeginInfoEXT`
- Each `sType` member in the `pNext` chain **must** be unique
- `renderPass` **must** be a valid `VkRenderPass` handle
- `framebuffer` **must** be a valid `VkFramebuffer` handle
- If `clearValueCount` is not `0`, `pClearValues` **must** be a valid pointer to an array of `clearValueCount` `VkClearValue` unions
- Both of `framebuffer`, and `renderPass` **must** have been created, allocated, or retrieved from the same `VkDevice`

The image layout of the depth aspect of a depth/stencil attachment referring to an image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` is dependent on the last sample locations used to render to the image subresource, thus preserving the contents of such depth/stencil attachments across subpass boundaries requires the application to specify these sample locations whenever a layout transition of the attachment **may** occur. This information **can** be provided by chaining an instance of the `VkRenderPassSampleLocationsBeginInfoEXT` structure to the `pNext` chain of `VkRenderPassBeginInfo`.

The `VkRenderPassSampleLocationsBeginInfoEXT` structure is defined as:

```
typedef struct VkRenderPassSampleLocationsBeginInfoEXT {
    VkStructureType           sType;
    const void*               pNext;
    uint32_t                  attachmentInitialSampleLocationsCount;
    const VkAttachmentSampleLocationsEXT* pAttachmentInitialSampleLocations;
    uint32_t                  postSubpassSampleLocationsCount;
    const VkSubpassSampleLocationsEXT* pPostSubpassSampleLocations;
} VkRenderPassSampleLocationsBeginInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `attachmentInitialSampleLocationsCount` is the number of elements in the `pAttachmentInitialSampleLocations` array.
- `pAttachmentInitialSampleLocations` is a pointer to an array of `VkAttachmentSampleLocationsEXT` structures specifying the attachment indices and their corresponding sample location state. Each element of `pAttachmentInitialSampleLocations` **can** specify the sample location state to use in the automatic layout transition performed to transition a depth/stencil attachment from the initial layout of the attachment to the image layout specified for the attachment in the first subpass using it.
- `postSubpassSampleLocationsCount` is the number of elements in the `pPostSubpassSampleLocations` array.
- `pPostSubpassSampleLocations` is a pointer to an array of `postSubpassSampleLocationsCount` `VkSubpassSampleLocationsEXT` structures specifying the subpass indices and their corresponding sample location state. Each element of `pPostSubpassSampleLocations` **can** specify the sample location state to use in the automatic layout transition performed to transition the depth/stencil attachment used by the specified subpass to the image layout specified in a dependent subpass or to the final layout of the attachment in case the specified subpass is the last subpass using that attachment. In addition, if `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE`, each element of `pPostSubpassSampleLocations` **must** specify the sample location state that matches the sample locations used by all pipelines that will be bound to a command buffer during the specified subpass. If `variableSampleLocations` is `VK_TRUE`, the sample locations used for rasterization do not depend on `pPostSubpassSampleLocations`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT`
- If `attachmentInitialSampleLocationsCount` is not `0`, `pAttachmentInitialSampleLocations` **must** be a valid pointer to an array of `attachmentInitialSampleLocationsCount` valid `VkAttachmentSampleLocationsEXT` structures
- If `postSubpassSampleLocationsCount` is not `0`, `pPostSubpassSampleLocations` **must** be a valid pointer to an array of `postSubpassSampleLocationsCount` valid `VkSubpassSampleLocationsEXT` structures

The `VkAttachmentSampleLocationsEXT` structure is defined as:

```
typedef struct VkAttachmentSampleLocationsEXT {  
    uint32_t attachmentIndex;  
    VkSampleLocationsInfoEXT sampleLocationsInfo;  
} VkAttachmentSampleLocationsEXT;
```

- `attachmentIndex` is the index of the attachment for which the sample locations state is provided.
- `sampleLocationsInfo` is the sample locations state to use for the layout transition of the given attachment from the initial layout of the attachment to the image layout specified for the attachment in the first subpass using it.

If the image referenced by the framebuffer attachment at index `attachmentIndex` was not created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` then the values specified in `sampleLocationsInfo` are ignored.

## Valid Usage

- `attachmentIndex` **must** be less than the `attachmentCount` specified in `VkRenderPassCreateInfo` the render pass specified by `VkRenderPassBeginInfo::renderPass` was created with

## Valid Usage (Implicit)

- `sampleLocationsInfo` **must** be a valid `VkSampleLocationsInfoEXT` structure

The `VkSubpassSampleLocationsEXT` structure is defined as:

```
typedef struct VkSubpassSampleLocationsEXT {  
    uint32_t subpassIndex;  
    VkSampleLocationsInfoEXT sampleLocationsInfo;  
} VkSubpassSampleLocationsEXT;
```

- `subpassIndex` is the index of the subpass for which the sample locations state is provided.
- `sampleLocationsInfo` is the sample locations state to use for the layout transition of the depth/stencil attachment away from the image layout the attachment is used with in the subpass specified in `subpassIndex`.

If the image referenced by the depth/stencil attachment used in the subpass identified by `subpassIndex` was not created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` or if the subpass does not use a depth/stencil attachment, and `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_TRUE` then the values specified in `sampleLocationsInfo` are ignored.

## Valid Usage

- `subpassIndex` **must** be less than the `subpassCount` specified in `VkRenderPassCreateInfo` the render pass specified by `VkRenderPassBeginInfo::renderPass` was created with

## Valid Usage (Implicit)

- `sampleLocationsInfo` **must** be a valid `VkSampleLocationsInfoEXT` structure

The `VkSubpassBeginInfoKHR` structure is defined as:

```
typedef struct VkSubpassBeginInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSubpassContents  contents;
} VkSubpassBeginInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `contents` is a `VkSubpassContents` value specifying how the commands in the next subpass will be provided.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO_KHR`
- `pNext` **must** be `NULL`
- `contents` **must** be a valid `VkSubpassContents` value

Possible values of `vkCmdBeginRenderPass::contents`, specifying how the commands in the first subpass will be provided, are:

```

typedef enum VkSubpassContents {
    VK_SUBPASS_CONTENTS_INLINE = 0,
    VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS = 1,
    VK_SUBPASS_CONTENTS_MAX_ENUM = 0x7FFFFFFF
} VkSubpassContents;

```

- **VK\_SUBPASS\_CONTENTS\_INLINE** specifies that the contents of the subpass will be recorded inline in the primary command buffer, and secondary command buffers **must** not be executed within the subpass.
- **VK\_SUBPASS\_CONTENTS\_SECONDARY\_COMMAND\_BUFFERS** specifies that the contents are recorded in secondary command buffers that will be called from the primary command buffer, and `vkCmdExecuteCommands` is the only valid command on the command buffer until `vkCmdNextSubpass` or `vkCmdEndRenderPass`.

If the `pNext` chain of `VkRenderPassBeginInfo` includes a `VkDeviceGroupRenderPassBeginInfo` structure, then that structure includes a device mask and set of render areas for the render pass instance.

The `VkDeviceGroupRenderPassBeginInfo` structure is defined as:

```

typedef struct VkDeviceGroupRenderPassBeginInfo {
    VkStructureType      sType;
    const void*        pNext;
    uint32_t             deviceMask;
    uint32_t             deviceRenderAreaCount;
    const VkRect2D*    pDeviceRenderAreas;
} VkDeviceGroupRenderPassBeginInfo;

```

or the equivalent

```
typedef VkDeviceGroupRenderPassBeginInfo VkDeviceGroupRenderPassBeginInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `deviceMask` is the device mask for the render pass instance.
- `deviceRenderAreaCount` is the number of elements in the `pDeviceRenderAreas` array.
- `pDeviceRenderAreas` is a pointer to an array of `VkRect2D` structures defining the render area for each physical device.

The `deviceMask` serves several purposes. It is an upper bound on the set of physical devices that **can** be used during the render pass instance, and the initial device mask when the render pass instance begins. In addition, commands transitioning to the next subpass in the render pass instance and commands ending the render pass instance, and, accordingly render pass attachment load, store, and resolve operations and subpass dependencies corresponding to the render pass instance, are executed on the physical devices included in the device mask provided here.

If `deviceRenderAreaCount` is not zero, then the elements of `pDeviceRenderAreas` override the value of `VkRenderPassBeginInfo::renderArea`, and provide a render area specific to each physical device. These render areas serve the same purpose as `VkRenderPassBeginInfo::renderArea`, including controlling the region of attachments that are cleared by `VK_ATTACHMENT_LOAD_OP_CLEAR` and that are resolved into resolve attachments.

If this structure is not present, the render pass instance's device mask is the value of `VkDeviceGroupCommandBufferBeginInfo::deviceMask`. If this structure is not present or if `deviceRenderAreaCount` is zero, `VkRenderPassBeginInfo::renderArea` is used for all physical devices.

## Valid Usage

- `deviceMask` **must** be a valid device mask value
- `deviceMask` **must** not be zero
- `deviceMask` **must** be a subset of the command buffer's initial device mask
- `deviceRenderAreaCount` **must** either be zero or equal to the number of physical devices in the logical device.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO`
- If `deviceRenderAreaCount` is not `0`, `pDeviceRenderAreas` **must** be a valid pointer to an array of `deviceRenderAreaCount` `VkRect2D` structures

The `VkRenderPassAttachmentBeginInfoKHR` structure is defined as:

```
typedef struct VkRenderPassAttachmentBeginInfoKHR {
    VkStructureType      sType;
    const void*          pNext;
    uint32_t             attachmentCount;
    const VkImageView*   pAttachments;
} VkRenderPassAttachmentBeginInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `attachmentCount` is the number of attachments.
- `pAttachments` is a pointer to an array of `VkImageView` handles, each of which will be used as the corresponding attachment in the render pass instance.

## Valid Usage

- Each element of `pAttachments` **must** only specify a single mip level
- Each element of `pAttachments` **must** have been created with the identity swizzle

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO_KHR`
- If `attachmentCount` is not `0`, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkImageView` handles

To query the render area granularity, call:

```
void vkGetRenderAreaGranularity(  
    VkDevice                                     device,  
    VkRenderPass                                renderPass,  
    VkExtent2D*                                  pGranularity);
```

- `device` is the logical device that owns the render pass.
- `renderPass` is a handle to a render pass.
- `pGranularity` is a pointer to a `VkExtent2D` structure in which the granularity is returned.

The conditions leading to an optimal `renderArea` are:

- the `offset.x` member in `renderArea` is a multiple of the `width` member of the returned `VkExtent2D` (the horizontal granularity).
- the `offset.y` member in `renderArea` is a multiple of the `height` of the returned `VkExtent2D` (the vertical granularity).
- either the `offset.width` member in `renderArea` is a multiple of the horizontal granularity or `offset.x+width` is equal to the `width` of the `framebuffer` in the `VkRenderPassBeginInfo`.
- either the `offset.height` member in `renderArea` is a multiple of the vertical granularity or `offset.y+height` is equal to the `height` of the `framebuffer` in the `VkRenderPassBeginInfo`.

Subpass dependencies are not affected by the render area, and apply to the entire image subresources attached to the framebuffer as specified in the description of [automatic layout transitions](#). Similarly, pipeline barriers are valid even if their effect extends outside the render area.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `renderPass` **must** be a valid `VkRenderPass` handle
- `pGranularity` **must** be a valid pointer to a `VkExtent2D` structure
- `renderPass` **must** have been created, allocated, or retrieved from `device`

To transition to the next subpass in the render pass instance after recording the commands for a subpass, call:

```
void vkCmdNextSubpass(  
    VkCommandBuffer  
    commandBuffer,  
    VkSubpassContents  
    contents);
```

- `commandBuffer` is the command buffer in which to record the command.
- `contents` specifies how the commands in the next subpass will be provided, in the same fashion as the corresponding parameter of `vkCmdBeginRenderPass`.

The subpass index for a render pass begins at zero when `vkCmdBeginRenderPass` is recorded, and increments each time `vkCmdNextSubpass` is recorded.

Moving to the next subpass automatically performs any multisample resolve operations in the subpass being ended. End-of-subpass multisample resolves are treated as color attachment writes for the purposes of synchronization. This applies to resolve operations for both color and depth/stencil attachments. That is, they are considered to execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage and their writes are synchronized with `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`. Synchronization between rendering within a subpass and any resolve operations at the end of the subpass occurs automatically, without need for explicit dependencies or pipeline barriers. However, if the resolve attachment is also used in a different subpass, an explicit dependency is needed.

After transitioning to the next subpass, the application **can** record the commands for that subpass.

## Valid Usage

- The current subpass index **must** be less than the number of subpasses in the render pass minus one
- This command **must** not be recorded when transform feedback is active

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `contents` **must** be a valid `VkSubpassContents` value
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- `commandBuffer` **must** be a primary `VkCommandBuffer`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Inside	Graphics	Graphics

To transition to the next subpass in the render pass instance after recording the commands for a subpass, call:

```
void vkCmdNextSubpass2KHR(  
    VkCommandBuffer  
    const VkSubpassBeginInfoKHR*  
    const VkSubpassEndInfoKHR*  
                                commandBuffer,  
                                pSubpassBeginInfo,  
                                pSubpassEndInfo);
```

- `commandBuffer` is the command buffer in which to record the command.
- `pSubpassBeginInfo` is a pointer to a `VkSubpassBeginInfoKHR` structure containing information about the subpass which is about to begin rendering.
- `pSubpassEndInfo` is a pointer to a `VkSubpassEndInfoKHR` structure containing information about how the previous subpass will be ended.

`vkCmdNextSubpass2KHR` is semantically identical to `vkCmdNextSubpass`, except that it is extensible, and that `contents` is provided as part of an extensible structure instead of as a flat parameter.

## Valid Usage

- The current subpass index **must** be less than the number of subpasses in the render pass minus one
- This command **must** not be recorded when transform feedback is active

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pSubpassBeginInfo` **must** be a valid pointer to a valid `VkSubpassBeginInfoKHR` structure
- `pSubpassEndInfo` **must** be a valid pointer to a valid `VkSubpassEndInfoKHR` structure
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- `commandBuffer` **must** be a primary `VkCommandBuffer`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Inside	Graphics	Graphics

To record a command to end a render pass instance after recording the commands for the last subpass, call:

```
void vkCmdEndRenderPass(  
    VkCommandBuffer  
        commandBuffer);
```

- `commandBuffer` is the command buffer in which to end the current render pass instance.

Ending a render pass instance performs any multisample resolve operations on the final subpass.

## Valid Usage

- The current subpass index **must** be equal to the number of subpasses in the render pass minus one
- This command **must** not be recorded when transform feedback is active

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- `commandBuffer` **must** be a primary `VkCommandBuffer`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Inside	Graphics	Graphics

To record a command to end a render pass instance after recording the commands for the last subpass, call:

```
void vkCmdEndRenderPass2KHR(  
    VkCommandBuffer  
    const VkSubpassEndInfoKHR*  
                                commandBuffer,  
                                pSubpassEndInfo);
```

- `commandBuffer` is the command buffer in which to end the current render pass instance.
- `pSubpassEndInfo` is a pointer to a `VkSubpassEndInfoKHR` structure containing information about how the previous subpass will be ended.

`vkCmdEndRenderPass2KHR` is semantically identical to `vkCmdEndRenderPass`, except that it is extensible.

## Valid Usage

- The current subpass index **must** be equal to the number of subpasses in the render pass minus one
- This command **must** not be recorded when transform feedback is active

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pSubpassEndInfo` **must** be a valid pointer to a valid `VkSubpassEndInfoKHR` structure
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- `commandBuffer` **must** be a primary `VkCommandBuffer`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Inside	Graphics	Graphics

The `VkSubpassEndInfoKHR` structure is defined as:

```
typedef struct VkSubpassEndInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
} VkSubpassEndInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SUBPASS_END_INFO_KHR`
- `pNext` **must** be `NULL`

# Chapter 8. Shaders

A shader specifies programmable operations that execute for each vertex, control point, tessellated vertex, primitive, fragment, or workgroup in the corresponding stage(s) of the graphics and compute pipelines.

Graphics pipelines include vertex shader execution as a result of [primitive assembly](#), followed, if enabled, by tessellation control and evaluation shaders operating on [patches](#), geometry shaders, if enabled, operating on primitives, and fragment shaders, if present, operating on fragments generated by [Rasterization](#). In this specification, vertex, tessellation control, tessellation evaluation and geometry shaders are collectively referred to as vertex processing stages and occur in the logical pipeline before rasterization. The fragment shader occurs logically after rasterization.

Only the compute shader stage is included in a compute pipeline. Compute shaders operate on compute invocations in a workgroup.

Shaders **can** read from input variables, and read from and write to output variables. Input and output variables **can** be used to transfer data between shader stages, or to allow the shader to interact with values that exist in the execution environment. Similarly, the execution environment provides constants that describe capabilities.

Shader variables are associated with execution environment-provided inputs and outputs using *built-in* decorations in the shader. The available decorations for each stage are documented in the following subsections.

## 8.1. Shader Modules

*Shader modules* contain *shader code* and one or more entry points. Shaders are selected from a shader module by specifying an entry point as part of [pipeline](#) creation. The stages of a pipeline **can** use shaders that come from different modules. The shader code defining a shader module **must** be in the SPIR-V format, as described by the [Vulkan Environment for SPIR-V](#) appendix.

Shader modules are represented by [VkShaderModule](#) handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkShaderModule)
```

To create a shader module, call:

```
VkResult vkCreateShaderModule(  
    VkDevice                                     device,  
    const VkShaderModuleCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkShaderModule* pShaderModule);
```

- **device** is the logical device that creates the shader module.
- **pCreateInfo** is a pointer to a [VkShaderModuleCreateInfo](#) structure.

- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pShaderModule` is a pointer to a `VkShaderModule` handle in which the resulting shader module object is returned.

Once a shader module has been created, any entry points it contains **can** be used in pipeline shader stages as described in [Compute Pipelines](#) and [Graphics Pipelines](#).

If the shader stage fails to compile `VK_ERROR_INVALID_SHADER_NV` will be generated and the compile log will be reported back to the application by `VK_EXT_debug_report` if enabled.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkShaderModuleCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pShaderModule` **must** be a valid pointer to a `VkShaderModule` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_SHADER_NV`

The `VkShaderModuleCreateInfo` structure is defined as:

```
typedef struct VkShaderModuleCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkShaderModuleCreateFlags flags;
    size_t                    codeSize;
    const uint32_t*           pCode;
} VkShaderModuleCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `codeSize` is the size, in bytes, of the code pointed to by `pCode`.
- `pCode` is a pointer to code that is used to create the shader module. The type and format of the

code is determined from the content of the memory addressed by `pCode`.

## Valid Usage

- `codeSize` **must** be greater than 0
- If `pCode` is a pointer to SPIR-V code, `codeSize` **must** be a multiple of 4
- `pCode` **must** point to either valid SPIR-V code, formatted and packed as described by the [Khronos SPIR-V Specification](#) or valid GLSL code which **must** be written to the [GL\\_KHR\\_vulkan\\_gsls](#) extension specification
- If `pCode` is a pointer to SPIR-V code, that code **must** adhere to the validation rules described by the [Validation Rules within a Module](#) section of the [SPIR-V Environment](#) appendix
- If `pCode` is a pointer to GLSL code, it **must** be valid GLSL code written to the [GL\\_KHR\\_vulkan\\_gsls](#) GLSL extension specification
- `pCode` **must** declare the `Shader` capability for SPIR-V code
- `pCode` **must** not declare any capability that is not supported by the API, as described by the [Capabilities](#) section of the [SPIR-V Environment](#) appendix
- If `pCode` declares any of the capabilities listed as **optional** in the [SPIR-V Environment](#) appendix, the corresponding feature(s) **must** be enabled.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of [VkShaderModuleValidationCacheCreateInfoEXT](#)
- `flags` **must** be `0`
- `pCode` **must** be a valid pointer to an array of  $\frac{\text{codeSize}}{4}$  `uint32_t` values

```
typedef VkFlags VkShaderModuleCreateFlags;
```

`VkShaderModuleCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To use a `VkValidationCacheEXT` to cache shader validation results, add a `VkShaderModuleValidationCacheCreateInfoEXT` to the `pNext` chain of the `VkShaderModuleCreateInfo` structure, specifying the cache object to use.

The `VkShaderModuleValidationCacheCreateInfoEXT` struct is defined as:

```
typedef struct VkShaderModuleValidationCacheCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkValidationCacheEXT validationCache;
} VkShaderModuleValidationCacheCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `validationCache` is the validation cache object from which the results of prior validation attempts will be written, and to which new validation results for this `VkShaderModule` will be written (if not already present).

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SHADER_MODULE_VALIDATION_CACHE_CREATE_INFO_EXT`
- `validationCache` **must** be a valid `VkValidationCacheEXT` handle

To destroy a shader module, call:

```
void vkDestroyShaderModule(
    VkDevice device,
    VkShaderModule shaderModule,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the shader module.
- `shaderModule` is the handle of the shader module to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

A shader module **can** be destroyed while pipelines created using its shaders are still in use.

## Valid Usage

- If `VkAllocationCallbacks` were provided when `shaderModule` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `shaderModule` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `shaderModule` is not `VK_NULL_HANDLE`, `shaderModule` **must** be a valid `VkShaderModule` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `shaderModule` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `shaderModule` **must** be externally synchronized

## 8.2. Shader Execution

At each stage of the pipeline, multiple invocations of a shader **may** execute simultaneously. Further, invocations of a single shader produced as the result of different commands **may** execute simultaneously. The relative execution order of invocations of the same shader type is undefined. Shader invocations **may** complete in a different order than that in which the primitives they originated from were drawn or dispatched by the application. However, fragment shader outputs are written to attachments in [rasterization order](#).

The relative execution order of invocations of different shader types is largely undefined. However, when invoking a shader whose inputs are generated from a previous pipeline stage, the shader invocations from the previous stage are guaranteed to have executed far enough to generate input values for all required inputs.

## 8.3. Shader Memory Access Ordering

The order in which image or buffer memory is read or written by shaders is largely undefined. For some shader types (vertex, tessellation evaluation, and in some cases, fragment), even the number of shader invocations that **may** perform loads and stores is undefined.

In particular, the following rules apply:

- [Vertex](#) and [tessellation evaluation](#) shaders will be invoked at least once for each unique vertex, as defined in those sections.
- [Fragment](#) shaders will be invoked zero or more times, as defined in that section.
- The relative execution order of invocations of the same shader type is undefined. A store issued by a shader when working on primitive B might complete prior to a store for primitive A, even if primitive A is specified prior to primitive B. This applies even to fragment shaders; while fragment shader outputs are always written to the framebuffer in [rasterization order](#), stores executed by fragment shader invocations are not.

- The relative execution order of invocations of different shader types is largely undefined.

*Note*



The above limitations on shader invocation order make some forms of synchronization between shader invocations within a single set of primitives unimplementable. For example, having one invocation poll memory written by another invocation assumes that the other invocation has been launched and will complete its writes in finite time.

The [Memory Model](#) appendix defines the terminology and rules for how to correctly communicate between shader invocations, such as when a write is [Visible-To](#) a read, and what constitutes a [Data Race](#).

Applications **must** not cause a data race.

## 8.4. Shader Inputs and Outputs

Data is passed into and out of shaders using variables with input or output storage class, respectively. User-defined inputs and outputs are connected between stages by matching their [Location](#) decorations. Additionally, data **can** be provided by or communicated to special functions provided by the execution environment using [BuiltIn](#) decorations.

In many cases, the same [BuiltIn](#) decoration **can** be used in multiple shader stages with similar meaning. The specific behavior of variables decorated as [BuiltIn](#) is documented in the following sections.

## 8.5. Task Shaders

Task shaders operate in conjunction with the mesh shaders to produce a collection of primitives that will be processed by subsequent stages of the graphics pipeline. Its primary purpose is to create a variable amount of subsequent mesh shader invocations.

Task shaders are invoked via the execution of the [programmable mesh shading](#) pipeline.

The task shader has no fixed-function inputs other than variables identifying the specific workgroup and invocation. The only fixed output of the task shader is a task count, identifying the number of mesh shader workgroups to create. The task shader can write additional outputs to task memory, which can be read by all of the mesh shader workgroups it created.

### 8.5.1. Task Shader Execution

Task workloads are formed from groups of work items called workgroups and processed by the task shader in the current graphics pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Task shaders execute in *global workgroups* which are divided into a number of *local workgroups* with a size that **can** be set by assigning a value to the [LocalSize](#) execution mode or via an object decorated by the [WorkgroupSize](#) decoration. An invocation within a local workgroup **can** share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members

of the local workgroup.

## 8.6. Mesh Shaders

Mesh shaders operate in workgroups to produce a collection of primitives that will be processed by subsequent stages of the graphics pipeline. Each workgroup emits zero or more output primitives and the group of vertices and their associated data required for each output primitive.

Mesh shaders are invoked via the execution of the [programmable mesh shading](#) pipeline.

The only inputs available to the mesh shader are variables identifying the specific workgroup and invocation and, if applicable, any outputs written to task memory by the task shader that spawned the mesh shader's workgroup. The mesh shader can operate without a task shader as well.

The invocations of the mesh shader workgroup write an output mesh, comprising a set of primitives with per-primitive attributes, a set of vertices with per-vertex attributes, and an array of indices identifying the mesh vertices that belong to each primitive. The primitives of this mesh are then processed by subsequent graphics pipeline stages, where the outputs of the mesh shader form an interface with the fragment shader.

### 8.6.1. Mesh Shader Execution

Mesh workloads are formed from groups of work items called workgroups and processed by the mesh shader in the current graphics pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Mesh shaders execute in *global workgroups* which are divided into a number of *local workgroups* with a size that **can** be set by assigning a value to the [LocalSize](#) execution mode or via an object decorated by the [WorkgroupSize](#) decoration. An invocation within a local workgroup **can** share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.

The *global workgroups* may be generated explicitly via the API, or implicitly through the task shader's work creation mechanism.

## 8.7. Vertex Shaders

Each vertex shader invocation operates on one vertex and its associated [vertex attribute](#) data, and outputs one vertex and associated data. Graphics pipelines using primitive shading **must** include a vertex shader, and the vertex shader stage is always the first shader stage in the graphics pipeline.

### 8.7.1. Vertex Shader Execution

A vertex shader **must** be executed at least once for each vertex specified by a draw command. If the subpass includes multiple views in its view mask, the shader **may** be invoked separately for each view. During execution, the shader is presented with the index of the vertex and instance for which it has been invoked. Input variables declared in the vertex shader are filled by the implementation with the values of vertex attributes associated with the invocation being executed.

If the same vertex is specified multiple times in a draw command (e.g. by including the same index value multiple times in an index buffer) the implementation **may** reuse the results of vertex shading if it can statically determine that the vertex shader invocations will produce identical results.

*Note*



It is implementation-dependent when and if results of vertex shading are reused, and thus how many times the vertex shader will be executed. This is true also if the vertex shader contains stores or atomic operations (see [vertexPipelineStoresAndAtomics](#)).

## 8.8. Tessellation Control Shaders

The tessellation control shader is used to read an input patch provided by the application and to produce an output patch. Each tessellation control shader invocation operates on an input patch (after all control points in the patch are processed by a vertex shader) and its associated data, and outputs a single control point of the output patch and its associated data, and **can** also output additional per-patch data. The input patch is sized according to the [patchControlPoints](#) member of [VkPipelineTessellationStateCreateInfo](#), as part of input assembly. The size of the output patch is controlled by the [OpExecutionMode OutputVertices](#) specified in the tessellation control or tessellation evaluation shaders, which **must** be specified in at least one of the shaders. The size of the input and output patches **must** each be greater than zero and less than or equal to [VkPhysicalDeviceLimits ::maxTessellationPatchSize](#).

### 8.8.1. Tessellation Control Shader Execution

A tessellation control shader is invoked at least once for each *output* vertex in a patch. If the subpass includes multiple views in its view mask, the shader **may** be invoked separately for each view.

Inputs to the tessellation control shader are generated by the vertex shader. Each invocation of the tessellation control shader **can** read the attributes of any incoming vertices and their associated data. The invocations corresponding to a given patch execute logically in parallel, with undefined relative execution order. However, the [OpControlBarrier](#) instruction **can** be used to provide limited control of the execution order by synchronizing invocations within a patch, effectively dividing tessellation control shader execution into a set of phases. Tessellation control shaders will read undefined values if one invocation reads a per-vertex or per-patch attribute written by another invocation at any point during the same phase, or if two invocations attempt to write different values to the same per-patch output in a single phase.

## 8.9. Tessellation Evaluation Shaders

The Tessellation Evaluation Shader operates on an input patch of control points and their associated data, and a single input barycentric coordinate indicating the invocation's relative position within the subdivided patch, and outputs a single vertex and its associated data.

### 8.9.1. Tessellation Evaluation Shader Execution

A tessellation evaluation shader is invoked at least once for each unique vertex generated by the tessellator. If the subpass includes multiple views in its view mask, the shader **may** be invoked separately for each view.

## 8.10. Geometry Shaders

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive.

### 8.10.1. Geometry Shader Execution

A geometry shader is invoked at least once for each primitive produced by the tessellation stages, or at least once for each primitive generated by [primitive assembly](#) when tessellation is not in use. A shader can request that the geometry shader runs multiple [instances](#). A geometry shader is invoked at least once for each instance. If the subpass includes multiple views in its view mask, the shader **may** be invoked separately for each view.

## 8.11. Fragment Shaders

Fragment shaders are invoked as the result of rasterization in a graphics pipeline. Each fragment shader invocation operates on a single fragment and its associated data. With few exceptions, fragment shaders do not have access to any data associated with other fragments and are considered to execute in isolation of fragment shader invocations associated with other fragments.

### 8.11.1. Fragment Shader Execution

For each fragment generated by rasterization, a fragment shader **may** be invoked. A fragment shader **must** not be invoked if the [Early Per-Fragment Tests](#) cause it to have no coverage. If the subpass includes multiple views in its view mask, the shader **may** be invoked separately for each view.

Furthermore, if it is determined that a fragment generated as the result of rasterizing a first primitive will have its outputs entirely overwritten by a fragment generated as the result of rasterizing a second primitive in the same subpass, and the fragment shader used for the fragment has no other side effects, then the fragment shader **may** not be executed for the fragment from the first primitive.

Relative ordering of execution of different fragment shader invocations is not defined.

For each fragment generated by a primitive, the number of times the fragment shader is invoked is implementation-dependent, but **must** obey the following constraints:

- Each covered sample is included in a single fragment shader invocation.
- When sample shading is not enabled, there is at least one fragment shader invocation.
- When sample shading is enabled, the minimum number of fragment shader invocations is as

defined in [Shading Rate Image](#) and [Sample Shading](#).

When there is more than one fragment shader invocation per fragment, the association of samples to invocations is implementation-dependent.

In addition to the conditions outlined above for the invocation of a fragment shader, a fragment shader invocation **may** be produced as a *helper invocation*. A helper invocation is a fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations. Stores and atomics performed by helper invocations **must** not have any effect on memory, and values returned by atomic instructions in helper invocations are undefined.

If the render pass has a fragment density map attachment, more than one fragment shader invocation **may** be invoked for each covered sample. Stores and atomics performed by these additional invocations have the normal effect. Such additional invocations are only produced if `VkPhysicalDeviceFragmentDensityMapPropertiesEXT::fragmentDensityInvocations` is `VK_TRUE`.

*Note*



Implementations **may** generate these additional fragment shader invocations in order to make transitions between fragment areas with different fragment densities more smooth.

### 8.11.2. Early Fragment Tests

An explicit control is provided to allow fragment shaders to enable early fragment tests. If the fragment shader specifies the `EarlyFragmentTests OpExecutionMode`, the per-fragment tests described in [Early Fragment Test Mode](#) are performed prior to fragment shader execution. Otherwise, they are performed after fragment shader execution.

If the fragment shader additionally specifies the `PostDepthCoverage OpExecutionMode`, the value of a variable decorated with the `SampleMask` built-in reflects the coverage after the early fragment tests. Otherwise, it reflects the coverage before the early fragment tests.

### 8.11.3. Fragment Shader Interlock

In normal operation, it is possible for more than one fragment shader invocation to be executed simultaneously for the same pixel if there are overlapping primitives. If the `fragmentShaderSampleInterlock`, `fragmentShaderPixelInterlock`, or `fragmentShaderShadingRateInterlock` features are enabled, it is possible to define a critical section within the fragment shader that is guaranteed to not run simultaneously with another fragment shader invocation for the same sample(s) or pixel(s). It is also possible to control the relative ordering of execution of these critical sections across different fragment shader invocations.

If the `FragmentShaderSampleInterlockEXT`, `FragmentShaderPixelInterlockEXT`, or `FragmentShaderShadingRateInterlockEXT` capabilities are declared in the fragment shader, the `OpBeginInvocationInterlockEXT` and `OpEndInvocationInterlockEXT` instructions **must** be used to delimit a critical section of fragment shader code.

To ensure each invocation of the critical section is executed in [primitive order](#), declare one of the

`PixelInterlockOrderedEXT`, `SampleInterlockOrderedEXT`, or `ShadingRateInterlockOrderedEXT` execution modes. If the order of execution of each invocation of the critical section does not matter, declare one of the `PixelInterlockUnorderedEXT`, `SampleInterlockUnorderedEXT`, or `ShadingRateInterlockUnorderedEXT` execution modes.

The `PixelInterlockOrderedEXT` and `PixelInterlockUnorderedEXT` execution modes provide mutual exclusion in the critical section for any pair of fragments corresponding to the same pixel, or pixels if the fragment covers more than one pixel. With sample shading enabled, these execution modes are treated like `SampleInterlockOrderedEXT` or `SampleInterlockUnorderedEXT` respectively.

The `SampleInterlockOrderedEXT` and `SampleInterlockUnorderedEXT` execution modes only provide mutual exclusion for pairs of fragments that both cover at least one common sample in the same pixel; these are recommended for performance if shaders use per-sample data structures. If these execution modes are used in single-sample mode they are treated like `PixelInterlockOrderedEXT` or `PixelInterlockUnorderedEXT` respectively.

The `ShadingRateInterlockOrderedEXT` and `ShadingRateInterlockUnorderedEXT` execution modes provide mutual exclusion for pairs of fragments that both have at least one common sample in the same pixel, even if none of the common samples are covered by both fragments. With sample shading enabled, these execution modes are treated like `SampleInterlockOrderedEXT` or `SampleInterlockUnorderedEXT` respectively.

## 8.12. Compute Shaders

Compute shaders are invoked via `vkCmdDispatch` and `vkCmdDispatchIndirect` commands. In general, they have access to similar resources as shader stages executing as part of a graphics pipeline.

Compute workloads are formed from groups of work items called workgroups and processed by the compute shader in the current compute pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Compute shaders execute in *global workgroups* which are divided into a number of *local workgroups* with a size that **can** be set by assigning a value to the `LocalSize` execution mode or via an object decorated by the `WorkgroupSize` decoration. An invocation within a local workgroup **can** share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.

## 8.13. Interpolation Decorations

Interpolation decorations control the behavior of attribute interpolation in the fragment shader stage. Interpolation decorations **can** be applied to `Input` storage class variables in the fragment shader stage's interface, and control the interpolation behavior of those variables.

Inputs that could be interpolated **can** be decorated by at most one of the following decorations:

- `Flat`: no interpolation
- `NoPerspective`: linear interpolation (for `lines` and `polygons`)

Fragment input variables decorated with neither `Flat` nor `NoPerspective` use perspective-correct interpolation (for `lines` and `polygons`).

The presence of and type of interpolation is controlled by the above interpolation decorations as well as the auxiliary decorations `Centroid` and `Sample`.

A variable decorated with `Flat` will not be interpolated. Instead, it will have the same value for every fragment within a triangle. This value will come from a single `provoking vertex`. A variable decorated with `Flat` **can** also be decorated with `Centroid` or `Sample`, which will mean the same thing as decorating it only as `Flat`.

For fragment shader input variables decorated with neither `Centroid` nor `Sample`, the assigned variable **may** be interpolated anywhere within the fragment and a single value **may** be assigned to each sample within the fragment.

If a fragment shader input is decorated with `Centroid`, a single value **may** be assigned to that variable for all samples in the fragment, but that value **must** be interpolated to a location that lies in both the fragment and in the primitive being rendered, including any of the fragment's samples covered by the primitive. Because the location at which the variable is interpolated **may** be different in neighboring fragments, and derivatives **may** be computed by computing differences between neighboring fragments, derivatives of centroid-sampled inputs **may** be less accurate than those for non-centroid interpolated variables. If

`VkPipelineViewportShadingRateImageStateCreateInfoNV::shadingRateImageEnable` is enabled, implementations **may** estimate derivatives using differencing without dividing by the distance between adjacent sample locations when the fragment size is larger than one pixel. The `PostDepthCoverage` execution mode does not affect the determination of the centroid location.

If a fragment shader input is decorated with `Sample`, a separate value **must** be assigned to that variable for each covered sample in the fragment, and that value **must** be sampled at the location of the individual sample. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment center **must** be used for `Centroid`, `Sample`, and undecorated attribute interpolation.

Fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type **must** be decorated with `Flat`.

When the `VK_AMD_shader_explicit_vertex_parameter` device extension is enabled inputs **can** be also decorated with the `CustomInterpAMD` interpolation decoration, including fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type. Inputs decorated with `CustomInterpAMD` **can** only be accessed by the extended instruction `InterpolateAtVertexAMD` and allows accessing the value of the input for individual vertices of the primitive.

When the `fragmentShaderBarycentric` feature is enabled, inputs **can** be also decorated with the `PerVertexNV` interpolation decoration, including fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type. Inputs decorated with `PerVertexNV` **can** only be accessed using an extra array dimension, where the extra index identifies one of the vertices of the primitive that produced the fragment.

## 8.14. Ray Generation Shaders

A ray generation shader is similar to a compute shader. Its main purpose is to execute ray tracing queries using `OpTraceNV` instructions and process the results.

### 8.14.1. Ray Generation Shader Execution

One ray generation shader is executed per ray tracing dispatch. Its location in the shader binding table (see [Shader Binding Table](#) for details) is passed directly into `vkCmdTraceRaysNV` using the `raygenShaderBindingTableBuffer` and `raygenShaderBindingOffset` parameters.

## 8.15. Intersection Shaders

Intersection shaders enable the implementation of arbitrary, application defined geometric primitives. An intersection shader for a primitive is executed whenever its axis-aligned bounding box is hit by a ray.

A built-in intersection shader for triangle primitives that is used automatically whenever geometry of type `VK_GEOMETRY_TYPE_TRIANGLES_NV` is specified.

Like other ray tracing shader domains, an intersection shader operates on a single ray at a time. It also operates on a single primitive at a time. It is therefore the purpose of an intersection shader to compute the ray-primitive intersections and report them. To report an intersection, the shader calls the `OpReportIntersectionNV` instruction.

An intersection shader communicates with any-hit and closest shaders by generating attribute values that they **can** read. Intersection shaders **cannot** read or modify the ray payload.

### 8.15.1. Intersection Shader Execution

The order in which intersections are found along a ray, and therefore the order in which intersection shaders are executed, is unspecified.

The intersection shader of the closest AABB which intersects the ray is guaranteed to be executed at some point during traversal, unless the ray is forcibly terminated.

## 8.16. Any-Hit Shaders

The any-hit shader is executed after the intersection shader reports an intersection that lies within the current [tmin,tmax] of the ray. The main use of any-hit shaders is to programmatically decide whether or not an intersection will be accepted. The intersection will be accepted unless the shader calls the `OpIgnoreIntersectionNV` instruction.

### 8.16.1. Any-Hit Shader Execution

The order in which intersections are found along a ray, and therefore the order in which any-hit shaders are executed, is unspecified.

The any-hit shader of the closest hit is guaranteed to be executed at some point during traversal, unless the ray is forcibly terminated.

## 8.17. Closest Hit Shaders

Closest hit shaders have read-only access to the attributes generated by the corresponding intersection shader, and **can** read or modify the ray payload. They also have access to a number of system-generated values. Closest hit shaders **can** call `OpTraceNV` to recursively trace rays.

### 8.17.1. Closest Hit Shader Execution

Exactly one closest hit shader is executed when traversal is finished and an intersection has been found and accepted.

## 8.18. Miss Shaders

Miss shaders **can** access the ray payload and **can** trace new rays through the `OpTraceNV` instruction, but **cannot** access attributes since they are not associated with an intersection.

### 8.18.1. Miss Shader Execution

A miss shader is executed instead of a closest hit shader if no intersection was found during traversal.

## 8.19. Callable Shaders

Callable shaders **can** access a callable payload that works similarly to ray payloads to do subroutine work.

### 8.19.1. Callable Shader Execution

A callable shader is executed by calling `OpExecuteCallableNV` from an allowed shader stage.

## 8.20. Static Use

A SPIR-V module declares a global object in memory using the `OpVariable` instruction, which results in a pointer `x` to that object. A specific entry point in a SPIR-V module is said to *statically use* that object if that entry point's call tree contains a function containing a memory instruction or image instruction with `x` as an `id` operand. See the “Memory Instructions” and “Image Instructions” subsections of section 3 “Binary Form” of the SPIR-V specification for the complete list of SPIR-V memory instructions.

Static use is not used to control the behavior of variables with `Input` and `Output` storage. The effects of those variables are applied based only on whether they are present in a shader entry point's interface.

## 8.21. Invocation and Derivative Groups

An *invocation group* (see the subsection “Control Flow” of section 2 of the SPIR-V specification) for a compute shader is the set of invocations in a single local workgroup. For graphics shaders, an invocation group is an implementation-dependent subset of the set of shader invocations of a given shader stage which are produced by a single drawing command. For indirect drawing commands with `drawCount` greater than one, invocations from separate draws are in distinct invocation groups.

*Note*



Because the partitioning of invocations into invocation groups is implementation-dependent and not observable, applications generally need to assume the worst case of all invocations in a draw belonging to a single invocation group.

A *derivative group* (see the subsection “Control Flow” of section 2 of the SPIR-V 1.00 Revision 4 specification) is a set of invocations which are used together to compute a derivative. For a fragment shader, a derivative group is generated by a single primitive (point, line, or triangle) and includes any helper invocations needed to compute derivatives. If the `subgroupSize` field of [VkPhysicalDeviceSubgroupProperties](#) is at least 4, a derivative group for a fragment shader corresponds to a single subgroup quad. Otherwise, a derivative group is the set of invocations generated by a single primitive. A derivative group for a compute shader is a single local workgroup.

Derivative values are undefined for a sampled image instruction if the instruction is in flow control that is not uniform across the derivative group.

## 8.22. Subgroups

A *subgroup* (see the subsection “Control Flow” of section 2 of the SPIR-V 1.3 Revision 1 specification) is a set of invocations that can synchronize and share data with each other efficiently. An invocation group is partitioned into one or more subgroups.

Subgroup operations are divided into various categories as described in [VkSubgroupFeatureFlagBits](#).

### 8.22.1. Basic Subgroup Operations

The basic subgroup operations allow two classes of functionality within shaders - elect and barrier. Invocations within a subgroup **can** choose a single invocation to perform some task for the subgroup as a whole using elect. Invocations within a subgroup **can** perform a subgroup barrier to ensure the ordering of execution or memory accesses within a subgroup. Barriers **can** be performed on buffer memory accesses, `WorkgroupLocal` memory accesses, and image memory accesses to ensure that any results written are visible by other invocations within the subgroup. An `OpControlBarrier` **can** also be used to perform a full execution control barrier. A full execution control barrier will ensure that each active invocation within the subgroup reaches a point of execution before any are allowed to continue.

## 8.22.2. Vote Subgroup Operations

The vote subgroup operations allow invocations within a subgroup to compare values across a subgroup. The types of votes enabled are:

- Do all active subgroup invocations agree that an expression is true?
- Do any active subgroup invocations evaluate an expression to true?
- Do all active subgroup invocations have the same value of an expression?

*Note*



These operations are useful in combination with control flow in that they allow for developers to check whether conditions match across the subgroup and choose potentially faster code-paths in these cases.

## 8.22.3. Arithmetic Subgroup Operations

The arithmetic subgroup operations allow invocations to perform scan and reduction operations across a subgroup. For reduction operations, each invocation in a subgroup will obtain the same result of these arithmetic operations applied across the subgroup. For scan operations, each invocation in the subgroup will perform an inclusive or exclusive scan, cumulatively applying the operation across the invocations in a subgroup in an implementation-defined order. The operations supported are add, mul, min, max, and, or, xor.

## 8.22.4. Ballot Subgroup Operations

The ballot subgroup operations allow invocations to perform more complex votes across the subgroup. The ballot functionality allows all invocations within a subgroup to provide a boolean value and get as a result what each invocation provided as their boolean value. The broadcast functionality allows values to be broadcast from an invocation to all other invocations within the subgroup, given that the invocation to be broadcast from is known at pipeline creation time.

## 8.22.5. Shuffle Subgroup Operations

The shuffle subgroup operations allow invocations to read values from other invocations within a subgroup.

## 8.22.6. Shuffle Relative Subgroup Operations

The shuffle relative subgroup operations allow invocations to read values from other invocations within the subgroup relative to the current invocation in the group. The relative operations supported allow data to be shifted up and down through the invocations within a subgroup.

## 8.22.7. Clustered Subgroup Operations

The clustered subgroup operations allow invocations to perform an operation among partitions of a subgroup, such that the operation is only performed within the subgroup invocations within a partition. The partitions for clustered subgroup operations are consecutive power-of-two size

groups of invocations and the cluster size **must** be known at pipeline creation time. The operations supported are add, mul, min, max, and, or, xor.

## 8.22.8. Quad Subgroup Operations

The quad subgroup operations allow clusters of 4 invocations (a quad), to share data efficiently with each other. For fragment shaders, if the `subgroupSize` field of `VkPhysicalDeviceSubgroupProperties` is at least 4, each quad corresponds to one of the groups of four shader invocations used for `derivatives`. For compute shaders using the `DerivativeGroupQuadsNV` or `DerivativeGroupLinearNV` execution modes, each quad corresponds to one of the groups of four shader invocations used for `derivatives`. The invocations in each quad are ordered to have attribute values of  $P_{i0,j0}$ ,  $P_{i1,j0}$ ,  $P_{i0,j1}$ , and  $P_{i1,j1}$ , respectively.

## 8.22.9. Partitioned Subgroup Operations

The partitioned subgroup operations allow a subgroup to partition its invocations into disjoint subsets and to perform scan and reduce operations among invocations belonging to the same subset. The partitions for partitioned subgroup operations are specified by a ballot operation and **can** be computed at runtime. The operations supported are add, mul, min, max, and, or, xor.

# 8.23. Cooperative Matrices

A *cooperative matrix* type is a SPIR-V type where the storage for and computations performed on the matrix are spread across a set of invocations such as a subgroup. These types give the implementation freedom in how to optimize matrix multiplies.

SPIR-V defines the types and instructions, but does not specify rules about what sizes/combinations are valid, and it is expected that different implementations **may** support different sizes.

To enumerate the supported cooperative matrix types and operations, call:

```
VkResult vkGetPhysicalDeviceCooperativeMatrixPropertiesNV(  
    VkPhysicalDevice           physicalDevice,  
    uint32_t*                  pPropertyCount,  
    VkCooperativeMatrixPropertiesNV* pProperties);
```

- `physicalDevice` is the physical device.
- `pPropertyCount` is a pointer to an integer related to the number of cooperative matrix properties available or queried.
- `pProperties` is either `NULL` or a pointer to an array of `VkCooperativeMatrixPropertiesNV` structures.

If `pProperties` is `NULL`, then the number of cooperative matrix properties available is returned in `pPropertyCount`. Otherwise, `pPropertyCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If `pPropertyCount` is less than the number of cooperative matrix properties available, at most `pPropertyCount` structures will be written. If `pPropertyCount` is

smaller than the number of cooperative matrix properties available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available cooperative matrix properties were returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkCooperativeMatrixPropertiesNV` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Each `VkCooperativeMatrixPropertiesNV` structure describes a single supported combination of types for a matrix multiply/add operation (`OpCooperativeMatrixMulAddNV`). The multiply **can** be described in terms of the following variables and types (in SPIR-V pseudocode):

```
%A is of type OpTypeCooperativeMatrixNV %AType %scope %MSize %KSize  
%B is of type OpTypeCooperativeMatrixNV %BType %scope %KSize %NSize  
%C is of type OpTypeCooperativeMatrixNV %CType %scope %MSize %NSize  
%D is of type OpTypeCooperativeMatrixNV %DType %scope %MSize %NSize  
  
%D = %A * %B + %C // using OpCooperativeMatrixMulAddNV
```

A matrix multiply with these dimensions is known as an  $M \times N \times K$  matrix multiply.

The `VkCooperativeMatrixPropertiesNV` structure is defined as:

```

typedef struct VkCooperativeMatrixPropertiesNV {
    VkStructureType      sType;
    void*              pNext;
    uint32_t             MSize;
    uint32_t             NSize;
    uint32_t             KSize;
    VkComponentTypeNV   AType;
    VkComponentTypeNV   BType;
    VkComponentTypeNV   CType;
    VkComponentTypeNV   DType;
    VkScopeNV            scope;
} VkCooperativeMatrixPropertiesNV;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **MSize** is the number of rows in matrices A, C, and D.
- **KSize** is the number of columns in matrix A and rows in matrix B.
- **NSize** is the number of columns in matrices B, C, D.
- **AType** is the component type of matrix A, of type [VkComponentTypeNV](#).
- **BType** is the component type of matrix B, of type [VkComponentTypeNV](#).
- **CType** is the component type of matrix C, of type [VkComponentTypeNV](#).
- **DType** is the component type of matrix D, of type [VkComponentTypeNV](#).
- **scope** is the scope of all the matrix types, of type [VkScopeNV](#).

If some types are preferred over other types (e.g. for performance), they **should** appear earlier in the list enumerated by [vkGetPhysicalDeviceCooperativeMatrixPropertiesNV](#).

At least one entry in the list **must** have power of two values for all of **MSize**, **KSize**, and **NSize**.

## Valid Usage (Implicit)

- **sType** **must** be [VK\\_STRUCTURE\\_TYPE\\_COOPERATIVE\\_MATRIX\\_PROPERTIES\\_NV](#)
- **pNext** **must** be **NULL**
- **AType** **must** be a valid [VkComponentTypeNV](#) value
- **BType** **must** be a valid [VkComponentTypeNV](#) value
- **CType** **must** be a valid [VkComponentTypeNV](#) value
- **DType** **must** be a valid [VkComponentTypeNV](#) value
- **scope** **must** be a valid [VkScopeNV](#) value

Possible values for [VkScopeNV](#) include:

```

typedef enum VkScopeNV {
    VK_SCOPE_DEVICE_NV = 1,
    VK_SCOPE_WORKGROUP_NV = 2,
    VK_SCOPE_SUBGROUP_NV = 3,
    VK_SCOPE_QUEUE_FAMILY_NV = 5,
    VK_SCOPE_MAX_ENUM_NV = 0x7FFFFFFF
} VkScopeNV;

```

- **VK\_SCOPE\_DEVICE\_NV** corresponds to SPIR-V **Device** scope.
- **VK\_SCOPE\_WORKGROUP\_NV** corresponds to SPIR-V **Workgroup** scope.
- **VK\_SCOPE\_SUBGROUP\_NV** corresponds to SPIR-V **Subgroup** scope.
- **VK\_SCOPE\_QUEUE\_FAMILY\_NV** corresponds to SPIR-V **QueueFamilyKHR** scope.

All enum values match the corresponding SPIR-V value.

Possible values for **VkComponentTypeNV** include:

```

typedef enum VkComponentTypeNV {
    VK_COMPONENT_TYPE_FLOAT16_NV = 0,
    VK_COMPONENT_TYPE_FLOAT32_NV = 1,
    VK_COMPONENT_TYPE_FLOAT64_NV = 2,
    VK_COMPONENT_TYPE_SINT8_NV = 3,
    VK_COMPONENT_TYPE_SINT16_NV = 4,
    VK_COMPONENT_TYPE_SINT32_NV = 5,
    VK_COMPONENT_TYPE_SINT64_NV = 6,
    VK_COMPONENT_TYPE_UINT8_NV = 7,
    VK_COMPONENT_TYPE_UINT16_NV = 8,
    VK_COMPONENT_TYPE_UINT32_NV = 9,
    VK_COMPONENT_TYPE_UINT64_NV = 10,
    VK_COMPONENT_TYPE_MAX_ENUM_NV = 0x7FFFFFFF
} VkComponentTypeNV;

```

- **VK\_COMPONENT\_TYPE\_FLOAT16\_NV** corresponds to SPIR-V **OpTypeFloat** 16.
- **VK\_COMPONENT\_TYPE\_FLOAT32\_NV** corresponds to SPIR-V **OpTypeFloat** 32.
- **VK\_COMPONENT\_TYPE\_FLOAT64\_NV** corresponds to SPIR-V **OpTypeFloat** 64.
- **VK\_COMPONENT\_TYPE\_SINT8\_NV** corresponds to SPIR-V **OpTypeInt** 8 1.
- **VK\_COMPONENT\_TYPE\_SINT16\_NV** corresponds to SPIR-V **OpTypeInt** 16 1.
- **VK\_COMPONENT\_TYPE\_SINT32\_NV** corresponds to SPIR-V **OpTypeInt** 32 1.
- **VK\_COMPONENT\_TYPE\_SINT64\_NV** corresponds to SPIR-V **OpTypeInt** 64 1.
- **VK\_COMPONENT\_TYPE\_UINT8\_NV** corresponds to SPIR-V **OpTypeInt** 8 0.
- **VK\_COMPONENT\_TYPE\_UINT16\_NV** corresponds to SPIR-V **OpTypeInt** 16 0.
- **VK\_COMPONENT\_TYPE\_UINT32\_NV** corresponds to SPIR-V **OpTypeInt** 32 0.

- `VK_COMPONENT_TYPE_UINT64_NV` corresponds to SPIR-V `OpTypeInt` 64 0.

## 8.24. Validation Cache

Validation cache objects allow the result of internal validation to be reused, both within a single application run and between multiple runs. Reuse within a single run is achieved by passing the same validation cache object when creating supported Vulkan objects. Reuse across runs of an application is achieved by retrieving validation cache contents in one run of an application, saving the contents, and using them to preinitialize a validation cache on a subsequent run. The contents of the validation cache objects are managed by the validation layers. Applications **can** manage the host memory consumed by a validation cache object and control the amount of data retrieved from a validation cache object.

Validation cache objects are represented by `VkValidationCacheEXT` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkValidationCacheEXT)
```

To create validation cache objects, call:

```
VkResult vkCreateValidationCacheEXT(
    VkDevice                                     device,
    const VkValidationCacheCreateInfoEXT*        pCreateInfo,
    const VkAllocationCallbacks*                 pAllocator,
    VkValidationCacheEXT*                      pValidationCache);
```

- `device` is the logical device that creates the validation cache object.
- `pCreateInfo` is a pointer to a `VkValidationCacheCreateInfoEXT` structure containing the initial parameters for the validation cache object.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pValidationCache` is a pointer to a `VkValidationCacheEXT` handle in which the resulting validation cache object is returned.

*Note*

Applications **can** track and manage the total host memory size of a validation cache object using the `pAllocator`. Applications **can** limit the amount of data retrieved from a validation cache object in `vkGetValidationCacheDataEXT`. Implementations **should** not internally limit the total number of entries added to a validation cache object or the total host memory consumed.



Once created, a validation cache **can** be passed to the `vkCreateShaderModule` command as part of the `VkShaderModuleCreateInfo pNext` chain. If a `VkShaderModuleValidationCacheCreateInfoEXT` object is part of the `VkShaderModuleCreateInfo::pNext` chain, and its `validationCache` field is not `VK_NULL_HANDLE`, the implementation will query it for possible reuse opportunities and update it with new content. The use of the validation cache object in these commands is internally synchronized, and the same validation cache object **can** be used in multiple threads simultaneously.

### Note



Implementations **should** make every effort to limit any critical sections to the actual accesses to the cache, which is expected to be significantly shorter than the duration of the `vkCreateShaderModule` command.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkValidationCacheCreateInfoEXT` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pValidationCache` **must** be a valid pointer to a `VkValidationCacheEXT` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkValidationCacheCreateInfoEXT` structure is defined as:

```
typedef struct VkValidationCacheCreateInfoEXT {
    VkStructureType           sType;
    const void*               pNext;
    VkValidationCacheCreateFlagsEXT flags;
    size_t                    initialDataSize;
    const void*               pInitialData;
} VkValidationCacheCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `initialDataSize` is the number of bytes in `pInitialData`. If `initialDataSize` is zero, the validation cache will initially be empty.
- `pInitialData` is a pointer to previously retrieved validation cache data. If the validation cache data is incompatible (as defined below) with the device, the validation cache will be initially empty. If `initialDataSize` is zero, `pInitialData` is ignored.

## Valid Usage

- If `initialDataSize` is not `0`, it **must** be equal to the size of `pInitialData`, as returned by `vkGetValidationCacheDataEXT` when `pInitialData` was originally retrieved
- If `initialDataSize` is not `0`, `pInitialData` **must** have been retrieved from a previous call to `vkGetValidationCacheDataEXT`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_VALIDATION_CACHE_CREATE_INFO_EXT`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- If `initialDataSize` is not `0`, `pInitialData` **must** be a valid pointer to an array of `initialDataSize` bytes

```
typedef VkFlags VkValidationCacheCreateFlagsEXT;
```

`VkValidationCacheCreateFlagsEXT` is a bitmask type for setting a mask, but is currently reserved for future use.

Validation cache objects **can** be merged using the command:

```
VkResult vkMergeValidationCachesEXT(  
    VkDevice                                     device,  
    VkValidationCacheEXT                         dstCache,  
    uint32_t                                      srcCacheCount,  
    const VkValidationCacheEXT*                  pSrcCaches);
```

- `device` is the logical device that owns the validation cache objects.
- `dstCache` is the handle of the validation cache to merge results into.
- `srcCacheCount` is the length of the `pSrcCaches` array.
- `pSrcCaches` is a pointer to an array of validation cache handles, which will be merged into `dstCache`. The previous contents of `dstCache` are included after the merge.

### Note



The details of the merge operation are implementation dependent, but implementations **should** merge the contents of the specified validation caches and prune duplicate entries.

## Valid Usage

- `dstCache` **must** not appear in the list of source caches

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `dstCache` **must** be a valid `VkValidationCacheEXT` handle
- `pSrcCaches` **must** be a valid pointer to an array of `srcCacheCount` valid `VkValidationCacheEXT` handles
- `srcCacheCount` **must** be greater than 0
- `dstCache` **must** have been created, allocated, or retrieved from `device`
- Each element of `pSrcCaches` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `dstCache` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Data **can** be retrieved from a validation cache object using the command:

```
VkResult vkGetValidationCacheDataEXT(  
    VkDevice                                     device,  
    VkValidationCacheEXT                         validationCache,  
    size_t*                                       pDatasize,  
    void*                                         pData);
```

- `device` is the logical device that owns the validation cache.
- `validationCache` is the validation cache to retrieve data from.
- `pDatasize` is a pointer to a value related to the amount of data in the validation cache, as described below.
- `pData` is either `NULL` or a pointer to a buffer.

If `pData` is `NULL`, then the maximum size of the data that **can** be retrieved from the validation cache, in bytes, is returned in `pDataSize`. Otherwise, `pDataSize` **must** point to a variable set by the user to the size of the buffer, in bytes, pointed to by `pData`, and on return the variable is overwritten with the amount of data actually written to `pData`.

If `pDataSize` is less than the maximum size that **can** be retrieved by the validation cache, at most `pDataSize` bytes will be written to `pData`, and `vkGetValidationCacheDataEXT` will return `VK_INCOMPLETE`. Any data written to `pData` is valid and **can** be provided as the `pInitialData` member of the `VkValidationCacheCreateInfoEXT` structure passed to `vkCreateValidationCacheEXT`.

Two calls to `vkGetValidationCacheDataEXT` with the same parameters **must** retrieve the same data unless a command that modifies the contents of the cache is called between them.

Applications **can** store the data retrieved from the validation cache, and use these data, possibly in a future run of the application, to populate new validation cache objects. The results of validation, however, **may** depend on the vendor ID, device ID, driver version, and other details of the device. To enable applications to detect when previously retrieved data is incompatible with the device, the initial bytes written to `pData` **must** be a header consisting of the following members:

*Table 11. Layout for validation cache header version VK\_VALIDATION\_CACHE\_HEADER\_VERSION\_ONE\_EXT*

Offset	Size	Meaning
0	4	length in bytes of the entire validation cache header written as a stream of bytes, with the least significant byte first
4	4	a <code>VkValidationCacheHeaderVersionEXT</code> value written as a stream of bytes, with the least significant byte first
8	<code>VK_UUID_SIZE</code>	a layer commit ID expressed as a UUID, which uniquely identifies the version of the validation layers used to generate these validation results

The first four bytes encode the length of the entire validation cache header, in bytes. This value includes all fields in the header including the validation cache version field and the size of the length field.

The next four bytes encode the validation cache version, as described for `VkValidationCacheHeaderVersionEXT`. A consumer of the validation cache **should** use the cache version to interpret the remainder of the cache header.

If `pDataSize` is less than what is necessary to store this header, nothing will be written to `pData` and zero will be written to `pDataSize`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `validationCache` **must** be a valid `VkValidationCacheEXT` handle
- `pDataSize` **must** be a valid pointer to a `size_t` value
- If the value referenced by `pDataSize` is not `0`, and `pData` is not `NULL`, `pData` **must** be a valid pointer to an array of `pDataSize` bytes
- `validationCache` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Possible values of the second group of four bytes in the header returned by `vkGetValidationCacheDataEXT`, encoding the validation cache version, are:

```
typedef enum VkValidationCacheHeaderVersionEXT {
    VK_VALIDATION_CACHE_HEADER_VERSION_ONE_EXT = 1,
    VK_VALIDATION_CACHE_HEADER_VERSION_MAX_ENUM_EXT = 0x7FFFFFFF
} VkValidationCacheHeaderVersionEXT;
```

- `VK_VALIDATION_CACHE_HEADER_VERSION_ONE_EXT` specifies version one of the validation cache.

To destroy a validation cache, call:

```
void vkDestroyValidationCacheEXT(
    VkDevice                                     device,
    VkValidationCacheEXT                         validationCache,
    const VkAllocationCallbacks*                  pAllocator);
```

- `device` is the logical device that destroys the validation cache object.
- `validationCache` is the handle of the validation cache to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- If `VkAllocationCallbacks` were provided when `validationCache` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `validationCache` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `validationCache` is not `VK_NULL_HANDLE`, `validationCache` **must** be a valid `VkValidationCacheEXT` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `validationCache` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `validationCache` **must** be externally synchronized

# Chapter 9. Pipelines

The following [figure](#) shows a block diagram of the Vulkan pipelines. Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a *graphics pipeline*, a *compute pipeline*, or a *ray tracing pipeline*.

The graphics pipeline can be operated in two modes, as either *primitive shading* or *mesh shading* pipeline.

## Primitive Shading

The first stage of the [graphics pipeline](#) ([Input Assembler](#)) assembles vertices to form geometric primitives such as points, lines, and triangles, based on a requested primitive topology. In the next stage ([Vertex Shader](#)) vertices **can** be transformed, computing positions and attributes for each vertex. If [tessellation](#) and/or [geometry](#) shaders are supported, they **can** then generate multiple primitives from a single input primitive, possibly changing the primitive topology or generating additional attribute data in the process.

## Mesh Shading

When using the [mesh shading](#) pipeline input primitives are not assembled implicitly, but explicitly through the ([Mesh Shader](#)). The work on the mesh pipeline is initiated by the application [drawing](#) a set of mesh tasks.

If an optional ([Task Shader](#)) is active, each task triggers the execution of a task shader workgroup that will generate a new set of tasks upon completion. Each of these spawned tasks, or each of the original dispatched tasks if no task shader is present, triggers the execution of a mesh shader workgroup that produces an output mesh with a variable-sized number of primitives assembled from vertices stored in the output mesh.

## Common

The final resulting primitives are [clipped](#) to a clip volume in preparation for the next stage, [Rasterization](#). The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or triangle. Each *fragment* so produced is fed to the next stage ([Fragment Shader](#)) that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect [depth buffering](#)), [blending](#) of incoming fragment colors with stored colors, as well as [masking](#), [stenciling](#), and other [logical operations](#) on fragment values.

Framebuffer operations read and write the color and depth/stencil attachments of the framebuffer for a given subpass of a [render pass instance](#). The attachments **can** be used as input attachments in the fragment shader in a later subpass of the same render pass.

The [compute pipeline](#) is a separate pipeline from the graphics pipeline, which operates on one-, two-, or three-dimensional workgroups which **can** read from and write to buffer and image

memory.

This ordering is meant only as a tool for describing Vulkan, not as a strict rule of how Vulkan is implemented, and we present it only as a means to organize the various operations of the pipelines. Actual ordering guarantees between pipeline stages are explained in detail in the [synchronization chapter](#).

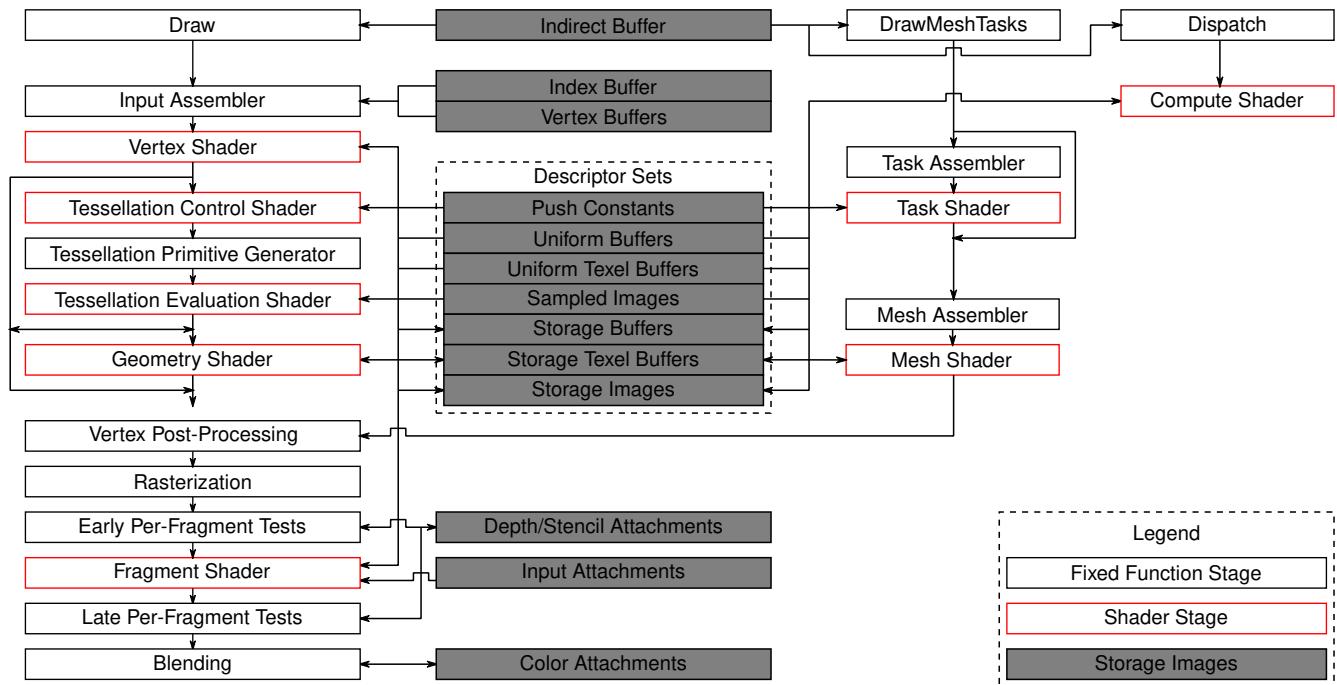


Figure 2. Block diagram of the Vulkan pipeline

Each pipeline is controlled by a monolithic object created from a description of all of the shader stages and any relevant fixed-function stages. [Linking](#) the whole pipeline together allows the optimization of shaders based on their input/outputs and eliminates expensive draw time state validation.

A pipeline object is bound to the current state using `vkCmdBindPipeline`. Any pipeline object state that is specified as `dynamic` is not applied to the current state when the pipeline object is bound, but is instead set by dynamic state setting commands.

No state, including dynamic state, is inherited from one command buffer to another.

Compute, graphics, and ray tracing pipelines are each represented by `VkPipeline` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipeline)
```

## 9.1. Compute Pipelines

Compute pipelines consist of a single static compute shader stage and the pipeline layout.

The compute pipeline represents a compute shader and is created by calling `vkCreateComputePipelines` with `module` and `pName` selecting an entry point from a shader module, where that entry point defines a valid compute shader, in the `VkPipelineShaderStageCreateInfo`

structure contained within the `VkComputePipelineCreateInfo` structure.

To create compute pipelines, call:

```
VkResult vkCreateComputePipelines(  
    VkDevice                                     device,  
    VkPipelineCache                            pipelineCache,  
    uint32_t                                    createInfoCount,  
    const VkComputePipelineCreateInfo*          pCreateInfos,  
    const VkAllocationCallbacks*                pAllocator,  
    VkPipeline*                                 pPipelines);
```

- `device` is the logical device that creates the compute pipelines.
- `pipelineCache` is either `VK_NULL_HANDLE`, indicating that pipeline caching is disabled; or the handle of a valid `pipeline cache` object, in which case use of that cache is enabled for the duration of the command.
- `createInfoCount` is the length of the `pCreateInfos` and `pPipelines` arrays.
- `pCreateInfos` is a pointer to an array of `VkComputePipelineCreateInfo` structures.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pPipelines` is a pointer to an array of `VkPipeline` handles in which the resulting compute pipeline objects are returned.

## Valid Usage

- If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and the `basePipelineIndex` member of that same element is not `-1`, `basePipelineIndex` **must** be less than the index into `pCreateInfos` that corresponds to that element
- If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, the base pipeline **must** have been created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` flag set

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `pipelineCache` is not `VK_NULL_HANDLE`, `pipelineCache` **must** be a valid `VkPipelineCache` handle
- `pCreateInfos` **must** be a valid pointer to an array of `createInfoCount` valid `VkComputePipelineCreateInfo` structures
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pPipelines` **must** be a valid pointer to an array of `createInfoCount` `VkPipeline` handles
- `createInfoCount` **must** be greater than `0`
- If `pipelineCache` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_SHADER_NV`

The `VkComputePipelineCreateInfo` structure is defined as:

```
typedef struct VkComputePipelineCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkPipelineCreateFlags       flags;
    VkPipelineShaderStageCreateInfo stage;
    VkPipelineLayout             layout;
    VkPipeline                  basePipelineHandle;
    int32_t                     basePipelineIndex;
} VkComputePipelineCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkPipelineCreateFlagBits` specifying how the pipeline will be generated.
- `stage` is a `VkPipelineShaderStageCreateInfo` structure describing the compute shader.
- `layout` is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.

- `basePipelineHandle` is a pipeline to derive from
- `basePipelineIndex` is an index into the `pCreateInfos` parameter to use as a pipeline to derive from

The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in [Pipeline Derivatives](#).

## Valid Usage

- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` **must** be a valid handle to a compute `VkPipeline`
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` **must** be a valid index into the calling command's `pCreateInfos` parameter
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not -1, `basePipelineHandle` **must** be `VK_NULL_HANDLE`
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` **must** be -1
- The `stage` member of `stage` **must** be `VK_SHADER_STAGE_COMPUTE_BIT`
- The shader code for the entry point identified by `stage` and the rest of the state identified by this structure **must** adhere to the pipeline linking rules described in the [Shader Interfaces](#) chapter
- `layout` **must** be consistent with the layout of the compute shader specified in `stage`
- The number of resources in `layout` accessible to the compute shader stage **must** be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkPipelineCompilerControlCreateInfoAMD` or `VkPipelineCreationFeedbackCreateInfoEXT`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be a valid combination of `VkPipelineCreateFlagBits` values
- `stage` **must** be a valid `VkPipelineShaderStageCreateInfo` structure
- `layout` **must** be a valid `VkPipelineLayout` handle
- Both of `basePipelineHandle`, and `layout` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

The `VkPipelineShaderStageCreateInfo` structure is defined as:

```
typedef struct VkPipelineShaderStageCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineShaderStageCreateFlags   flags;
    VkShaderStageFlagBits      stage;
    VkShaderModule              module;
    const char*                pName;
    const VkSpecializationInfo* pSpecializationInfo;
} VkPipelineShaderStageCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkPipelineShaderStageCreateFlagBits` specifying how the pipeline shader stage will be generated.
- `stage` is a `VkShaderStageFlagBits` value specifying a single pipeline stage.
- `module` is a `VkShaderModule` object containing the shader for this stage.
- `pName` is a pointer to a null-terminated UTF-8 string specifying the entry point name of the shader for this stage.
- `pSpecializationInfo` is a pointer to a `VkSpecializationInfo` structure, as described in [Specialization Constants](#), or `NULL`.

## Valid Usage

- If the `geometry shaders` feature is not enabled, `stage` **must** not be `VK_SHADER_STAGE_GEOMETRY_BIT`
- If the `tessellation shaders` feature is not enabled, `stage` **must** not be `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`
- If the `mesh shader` feature is not enabled, `stage` **must** not be `VK_SHADER_STAGE_MESH_BIT_NV`
- If the `task shader` feature is not enabled, `stage` **must** not be `VK_SHADER_STAGE_TASK_BIT_NV`
- `stage` **must** not be `VK_SHADER_STAGE_ALL_GRAPHICS`, or `VK_SHADER_STAGE_ALL`
- `pName` **must** be the name of an `OpEntryPoint` in `module` with an execution model that matches `stage`
- If the identified entry point includes any variable in its interface that is declared with the `ClipDistance BuiltIn` decoration, that variable **must** not have an array size greater than `VkPhysicalDeviceLimits::maxClipDistances`
- If the identified entry point includes any variable in its interface that is declared with the `CullDistance BuiltIn` decoration, that variable **must** not have an array size greater than `VkPhysicalDeviceLimits::maxCullDistances`
- If the identified entry point includes any variables in its interface that are declared with the `ClipDistance` or `CullDistance BuiltIn` decoration, those variables **must** not have array sizes which sum to more than `VkPhysicalDeviceLimits::maxCombinedClipAndCullDistances`
- If the identified entry point includes any variable in its interface that is declared with the `SampleMask BuiltIn` decoration, that variable **must** not have an array size greater than `VkPhysicalDeviceLimits::maxSampleMaskWords`
- If `stage` is `VK_SHADER_STAGE_VERTEX_BIT`, the identified entry point **must** not include any input variable in its interface that is decorated with `CullDistance`
- If `stage` is `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, and the identified entry point has an `OpExecutionMode` instruction that specifies a patch size with `OutputVertices`, the patch size **must** be greater than `0` and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`
- If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies a maximum output vertex count that is greater than `0` and less than or equal to `VkPhysicalDeviceLimits::maxGeometryOutputVertices`
- If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies an invocation count that is greater than `0` and less than or equal to `VkPhysicalDeviceLimits::maxGeometryShaderInvocations`
- If `stage` is a vertex processing stage, and the identified entry point writes to `Layer` for any primitive, it **must** write the same value to `Layer` for all vertices of a given primitive
- If `stage` is a vertex processing stage, and the identified entry point writes to `ViewportIndex` for any primitive, it **must** write the same value to `ViewportIndex` for all vertices of a given primitive

- If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, the identified entry point **must** not include any output variables in its interface decorated with `CullDistance`
- If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, and the identified entry point writes to `FragDepth` in any execution path, it **must** write to `FragDepth` in all execution paths
- If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, and the identified entry point writes to `FragStencilRefEXT` in any execution path, it **must** write to `FragStencilRefEXT` in all execution paths
- If `stage` is `VK_SHADER_STAGE_MESH_BIT_NV`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies a maximum output vertex count, `OutputVertices`, that is greater than `0` and less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputVertices`.
- If `stage` is `VK_SHADER_STAGE_MESH_BIT_NV`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies a maximum output primitive count, `OutputPrimitivesNV`, that is greater than `0` and less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputPrimitives`.
- If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set, the `subgroupSizeControl` feature **must** be enabled.
- If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` flag set, the `computeFullSubgroups` feature **must** be enabled.
- If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is chained to `pNext`, `flags` **must** not have the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set.
- If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is chained to `pNext`, the `subgroupSizeControl` feature **must** be enabled, and `stage` **must** be a valid bit specified in `requiredSubgroupSizeStages`.
- If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is chained to `pNext` and `stage` is `VK_SHADER_STAGE_COMPUTE_BIT` then local workgroup size of the shader **must** be less than or equal to the product of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT::requiredSubgroupSize` and `maxComputeWorkgroupSubgroups`.
- If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is chained to `pNext`, and `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` flag set, the local workgroup size in the X dimension of the pipeline **must** be a multiple of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT::requiredSubgroupSize`.
- If `flags` has both the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` and `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flags set, the local workgroup size in the X dimension of the pipeline **must** be a multiple of `maxSubgroupSize`.
- If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` flag set and `flags` does not have the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set and no `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is chained to `pNext`, the local workgroup size in the X dimension of the pipeline **must** be a multiple of

`subgroupSize`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT`
- `flags` **must** be a valid combination of `VkPipelineShaderStageCreateFlagBits` values
- `stage` **must** be a valid `VkShaderStageFlagBits` value
- `module` **must** be a valid `VkShaderModule` handle
- `pName` **must** be a null-terminated UTF-8 string
- If `pSpecializationInfo` is not `NULL`, `pSpecializationInfo` **must** be a valid pointer to a valid `VkSpecializationInfo` structure

```
typedef VkFlags VkPipelineShaderStageCreateFlags;
```

`VkPipelineShaderStageCreateFlags` is a bitmask type for setting a mask of zero or more `VkPipelineShaderStageCreateFlagBits`.

Possible values of the `flags` member of `VkPipelineShaderStageCreateInfo` specifying how a pipeline shader stage is created, are:

```
typedef enum VkPipelineShaderStageCreateFlagBits {
    VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT = 0x00000001,
    VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT = 0x00000002,
    VK_PIPELINE_SHADER_STAGE_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPipelineShaderStageCreateFlagBits;
```

- `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` specifies that the `SubgroupSize` **may** vary in the shader stage.
- `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` specifies that the subgroup sizes **must** be launched with all invocations active in the compute stage.

### Note

If `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` and `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` are specified and `minSubgroupSize` does not equal `maxSubgroupSize` and no required subgroup size is specified, then the only way to guarantee that the 'X' dimension of the local workgroup size is a multiple of `SubgroupSize` is to make it a multiple of `maxSubgroupSize`. Under these conditions, you are guaranteed full subgroups but not any particular subgroup size.



Commands and structures which need to specify one or more shader stages do so using a bitmask whose bits correspond to stages. Bits which **can** be set to specify shader stages are:

```
typedef enum VkShaderStageFlagBits {
    VK_SHADER_STAGE_VERTEX_BIT = 0x00000001,
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT = 0x00000002,
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT = 0x00000004,
    VK_SHADER_STAGE_GEOMETRY_BIT = 0x00000008,
    VK_SHADER_STAGE_FRAGMENT_BIT = 0x00000010,
    VK_SHADER_STAGE_COMPUTE_BIT = 0x00000020,
    VK_SHADER_STAGE_ALL_GRAPHICS = 0x0000001F,
    VK_SHADER_STAGE_ALL = 0x7FFFFFFF,
    VK_SHADER_STAGE_RAYGEN_BIT_NV = 0x00000100,
    VK_SHADER_STAGE_ANY_HIT_BIT_NV = 0x00000200,
    VK_SHADER_STAGE_CLOSEST_HIT_BIT_NV = 0x00000400,
    VK_SHADER_STAGE_MISS_BIT_NV = 0x00000800,
    VK_SHADER_STAGE_INTERSECTION_BIT_NV = 0x00001000,
    VK_SHADER_STAGE_CALLABLE_BIT_NV = 0x00002000,
    VK_SHADER_STAGE_TASK_BIT_NV = 0x00000040,
    VK_SHADER_STAGE_MESH_BIT_NV = 0x00000080,
    VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkShaderStageFlagBits;
```

- **VK\_SHADER\_STAGE\_VERTEX\_BIT** specifies the vertex stage.
- **VK\_SHADER\_STAGE\_TESSELLATION\_CONTROL\_BIT** specifies the tessellation control stage.
- **VK\_SHADER\_STAGE\_TESSELLATION\_EVALUATION\_BIT** specifies the tessellation evaluation stage.
- **VK\_SHADER\_STAGE\_GEOMETRY\_BIT** specifies the geometry stage.
- **VK\_SHADER\_STAGE\_FRAGMENT\_BIT** specifies the fragment stage.
- **VK\_SHADER\_STAGE\_COMPUTE\_BIT** specifies the compute stage.
- **VK\_SHADER\_STAGE\_TASK\_BIT\_NV** specifies the task stage.
- **VK\_SHADER\_STAGE\_MESH\_BIT\_NV** specifies the mesh stage.
- **VK\_SHADER\_STAGE\_ALL\_GRAPHICS** is a combination of bits used as shorthand to specify all graphics stages defined above (excluding the compute stage).
- **VK\_SHADER\_STAGE\_ALL** is a combination of bits used as shorthand to specify all shader stages supported by the device, including all additional stages which are introduced by extensions.
- **VK\_SHADER\_STAGE\_RAYGEN\_BIT\_NV** specifies the ray generation stage.
- **VK\_SHADER\_STAGE\_ANY\_HIT\_BIT\_NV** specifies the any-hit stage.
- **VK\_SHADER\_STAGE\_CLOSEST\_HIT\_BIT\_NV** specifies the closest hit stage.
- **VK\_SHADER\_STAGE\_MISS\_BIT\_NV** specifies the miss stage.
- **VK\_SHADER\_STAGE\_INTERSECTION\_BIT\_NV** specifies the intersection stage.
- **VK\_SHADER\_STAGE\_CALLABLE\_BIT\_NV** specifies the callable stage.

*Note*



`VK_SHADER_STAGE_ALL_GRAPHICS` only includes the original five graphics stages included in Vulkan 1.0, and not any stages added by extensions. Thus, it may not have the desired effect in all cases.

```
typedef VkFlags VkShaderStageFlags;
```

`VkShaderStageFlags` is a bitmask type for setting a mask of zero or more `VkShaderStageFlagBits`.

The `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is defined as:

```
typedef struct VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           requiredSubgroupSize;
} VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `requiredSubgroupSize` is an unsigned integer value that specifies the required subgroup size for the newly created pipeline shader stage.

### Valid Usage

- `requiredSubgroupSize` **must** be a power-of-two integer.
- `requiredSubgroupSize` **must** be greater or equal to `minSubgroupSize`.
- `requiredSubgroupSize` **must** be less than or equal to `maxSubgroupSize`.

If the `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is included in the `pNext` chain of `VkPipelineShaderStageCreateInfo`, it specifies that the pipeline shader stage being compiled has a required subgroup size.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO_EXT`

## 9.2. Graphics Pipelines

Graphics pipelines consist of multiple shader stages, multiple fixed-function pipeline stages, and a pipeline layout.

To create graphics pipelines, call:

```
VkResult vkCreateGraphicsPipelines(  
    VkDevice                                     device,  
    VkPipelineCache                            pipelineCache,  
    uint32_t                                    createInfoCount,  
    const VkGraphicsPipelineCreateInfo*        pCreateInfo,  
    const VkAllocationCallbacks*                pAllocator,  
    VkPipeline*                                pPipelines);
```

- `device` is the logical device that creates the graphics pipelines.
- `pipelineCache` is either `VK_NULL_HANDLE`, indicating that pipeline caching is disabled; or the handle of a valid `pipeline cache` object, in which case use of that cache is enabled for the duration of the command.
- `createInfoCount` is the length of the `pCreateInfos` and `pPipelines` arrays.
- `pCreateInfos` is a pointer to an array of `VkGraphicsPipelineCreateInfo` structures.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pPipelines` is a pointer to an array of `VkPipeline` handles in which the resulting graphics pipeline objects are returned.

The `VkGraphicsPipelineCreateInfo` structure includes an array of shader create info structures containing all the desired active shader stages, as well as creation info to define all relevant fixed-function stages, and a pipeline layout.

## Valid Usage

- If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and the `basePipelineIndex` member of that same element is not `-1`, `basePipelineIndex` **must** be less than the index into `pCreateInfos` that corresponds to that element
- If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, the base pipeline **must** have been created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` flag set

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `pipelineCache` is not `VK_NULL_HANDLE`, `pipelineCache` **must** be a valid `VkPipelineCache` handle
- `pCreateInfos` **must** be a valid pointer to an array of `createInfoCount` valid `VkGraphicsPipelineCreateInfo` structures
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pPipelines` **must** be a valid pointer to an array of `createInfoCount` `VkPipeline` handles
- `createInfoCount` **must** be greater than `0`
- If `pipelineCache` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_SHADER_NV`

The `VkGraphicsPipelineCreateInfo` structure is defined as:

```

typedef struct VkGraphicsPipelineCreateInfo {
    VkStructureType
    const void*
    VkPipelineCreateFlags
    uint32_t
    const VkPipelineShaderStageCreateInfo*
    const VkPipelineVertexInputStateCreateInfo*
    const VkPipelineInputAssemblyStateCreateInfo*
    const VkPipelineTessellationStateCreateInfo*
    const VkPipelineViewportStateCreateInfo*
    const VkPipelineRasterizationStateCreateInfo*
    const VkPipelineMultisampleStateCreateInfo*
    const VkPipelineDepthStencilStateCreateInfo*
    const VkPipelineColorBlendStateCreateInfo*
    const VkPipelineDynamicStateCreateInfo*
    VkPipelineLayout
    VkRenderPass
    uint32_t
    VkPipeline
    int32_t
} VkGraphicsPipelineCreateInfo;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is a bitmask of **VkPipelineCreateFlagBits** specifying how the pipeline will be generated.
- **stageCount** is the number of entries in the **pStages** array.
- **pStages** is a pointer to an array of **stageCount** **VkPipelineShaderStageCreateInfo** structures describing the set of the shader stages to be included in the graphics pipeline.
- **pVertexInputState** is a pointer to a **VkPipelineVertexInputStateCreateInfo** structure. It is ignored if the pipeline includes a mesh shader stage.
- **pInputAssemblyState** is a pointer to a **VkPipelineInputAssemblyStateCreateInfo** structure which determines input assembly behavior, as described in [Drawing Commands](#). It is ignored if the pipeline includes a mesh shader stage.
- **pTessellationState** is a pointer to a **VkPipelineTessellationStateCreateInfo** structure, and is ignored if the pipeline does not include a tessellation control shader stage and tessellation evaluation shader stage.
- **pViewportState** is a pointer to a **VkPipelineViewportStateCreateInfo** structure, and is ignored if the pipeline has rasterization disabled.
- **pRasterizationState** is a pointer to a **VkPipelineRasterizationStateCreateInfo** structure.
- **pMultisampleState** is a pointer to a **VkPipelineMultisampleStateCreateInfo** structure, and is ignored if the pipeline has rasterization disabled.
- **pDepthStencilState** is a pointer to a **VkPipelineDepthStencilStateCreateInfo** structure, and is ignored if the pipeline has rasterization disabled or if the subpass of the render pass the pipeline is created against does not use a depth/stencil attachment.

- `pColorBlendState` is a pointer to a `VkPipelineColorBlendStateCreateInfo` structure, and is ignored if the pipeline has rasterization disabled or if the subpass of the render pass the pipeline is created against does not use any color attachments.
- `pDynamicState` is a pointer to a `VkPipelineDynamicStateCreateInfo` structure, and is used to indicate which properties of the pipeline state object are dynamic and **can** be changed independently of the pipeline state. This **can** be `NULL`, which means no state in the pipeline is considered dynamic.
- `layout` is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
- `renderPass` is a handle to a render pass object describing the environment in which the pipeline will be used; the pipeline **must** only be used with an instance of any render pass compatible with the one provided. See [Render Pass Compatibility](#) for more information.
- `subpass` is the index of the subpass in the render pass where this pipeline will be used.
- `basePipelineHandle` is a pipeline to derive from.
- `basePipelineIndex` is an index into the `pCreateInfos` parameter to use as a pipeline to derive from.

The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in [Pipeline Derivatives](#).

If any shader stage fails to compile, the compile log will be reported back to the application, and `VK_ERROR_INVALID_SHADER_NV` will be generated.

## Valid Usage

- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` **must** be a valid handle to a graphics `VkPipeline`
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` **must** be a valid index into the calling command's `pCreateInfos` parameter
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not -1, `basePipelineHandle` **must** be `VK_NULL_HANDLE`
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` **must** be -1
- The `stage` member of each element of `pStages` **must** be unique
- The geometric shader stages provided in `pStages` **must** be either from the mesh shading pipeline (`stage` is `VK_SHADER_STAGE_TASK_BIT_NV` or `VK_SHADER_STAGE_MESH_BIT_NV`) or from the primitive shading pipeline (`stage` is `VK_SHADER_STAGE_VERTEX_BIT`, `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT`, `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, or `VK_SHADER_STAGE_GEOMETRY_BIT`).
- The `stage` member of one element of `pStages` **must** be either `VK_SHADER_STAGE_VERTEX_BIT` or `VK_SHADER_STAGE_MESH_BIT_NV`.
- The `stage` member of each element of `pStages` **must** not be `VK_SHADER_STAGE_COMPUTE_BIT`
- If `pStages` includes a tessellation control shader stage, it **must** include a tessellation evaluation shader stage
- If `pStages` includes a tessellation evaluation shader stage, it **must** include a tessellation control shader stage
- If `pStages` includes a tessellation control shader stage and a tessellation evaluation shader stage, `pTessellationState` **must** be a valid pointer to a valid `VkPipelineTessellationStateCreateInfo` structure
- If `pStages` includes tessellation shader stages, the shader code of at least one stage **must** contain an `OpExecutionMode` instruction that specifies the type of subdivision in the pipeline
- If `pStages` includes tessellation shader stages, and the shader code of both stages contain an `OpExecutionMode` instruction that specifies the type of subdivision in the pipeline, they **must** both specify the same subdivision mode
- If `pStages` includes tessellation shader stages, the shader code of at least one stage **must** contain an `OpExecutionMode` instruction that specifies the output patch size in the pipeline
- If `pStages` includes tessellation shader stages, and the shader code of both contain an `OpExecutionMode` instruction that specifies the out patch size in the pipeline, they **must** both specify the same patch size
- If `pStages` includes tessellation shader stages, the `topology` member of `pInputAssembly` **must** be `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`
- If the `topology` member of `pInputAssembly` is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, `pStages`

**must** include tessellation shader stages

- If `pStages` includes a geometry shader stage, and does not include any tessellation shader stages, its shader code **must** contain an `OpExecutionMode` instruction that specifies an input primitive type that is `compatible` with the primitive topology specified in `pInputAssembly`
- If `pStages` includes a geometry shader stage, and also includes tessellation shader stages, its shader code **must** contain an `OpExecutionMode` instruction that specifies an input primitive type that is `compatible` with the primitive topology that is output by the tessellation stages
- If `pStages` includes a fragment shader stage and a geometry shader stage, and the fragment shader code reads from an input variable that is decorated with `PrimitiveID`, then the geometry shader code **must** write to a matching output variable, decorated with `PrimitiveID`, in all execution paths
- If `pStages` includes a fragment shader stage, its shader code **must** not read from any input attachment that is defined as `VK_ATTACHMENT_UNUSED` in `subpass`
- The shader code for the entry points identified by `pStages`, and the rest of the state identified by this structure **must** adhere to the pipeline linking rules described in the [Shader Interfaces](#) chapter
- If rasterization is not disabled and `subpass` uses a depth/stencil attachment in `renderPass` that has a layout of `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` in the `VkAttachmentReference` defined by `subpass`, the `depthWriteEnable` member of `pDepthStencilState` **must** be `VK_FALSE`
- If rasterization is not disabled and `subpass` uses a depth/stencil attachment in `renderPass` that has a layout of `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` in the `VkAttachmentReference` defined by `subpass`, the `failOp`, `passOp` and `depthFailOp` members of each of the `front` and `back` members of `pDepthStencilState` **must** be `VK_STENCIL_OP_KEEP`
- If rasterization is not disabled and the subpass uses color attachments, then for each color attachment in the subpass the `blendEnable` member of the corresponding element of the `pAttachment` member of `pColorBlendState` **must** be `VK_FALSE` if the attached image's `format features` does not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`.
- If rasterization is not disabled and the subpass uses color attachments, the `attachmentCount` member of `pColorBlendState` **must** be equal to the `colorAttachmentCount` used to create `subpass`
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_VIEWPORT`, the `pViewports` member of `pViewportState` **must** be a valid pointer to an array of `pViewportState::viewportCount` valid `VkViewport` structures
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SCISSOR`, the `pScissors` member of `pViewportState` **must** be a valid pointer to an array of `pViewportState::scissorCount` `VkRect2D` structures
- If the wide lines feature is not enabled, and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_LINE_WIDTH`, the `lineWidth` member of `pRasterizationState` **must** be `1.0`
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, `pViewportState`

**must** be a valid pointer to a valid `VkPipelineViewportStateCreateInfo` structure

- If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, `pMultisampleState` **must** be a valid pointer to a valid `VkPipelineMultisampleStateCreateInfo` structure
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, and `subpass` uses a depth/stencil attachment, `pDepthStencilState` **must** be a valid pointer to a valid `VkPipelineDepthStencilStateCreateInfo` structure
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, and `subpass` uses color attachments, `pColorBlendState` **must** be a valid pointer to a valid `VkPipelineColorBlendStateCreateInfo` structure
- If the depth bias clamping feature is not enabled, no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DEPTH_BIAS`, and the `depthBiasEnable` member of `pRasterizationState` is `VK_TRUE`, the `depthBiasClamp` member of `pRasterizationState` **must** be `0.0`
- If the `VK_EXT_depth_range_unrestricted` extension is not enabled and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DEPTH_BOUNDS`, and the `depthBoundsTestEnable` member of `pDepthStencilState` is `VK_TRUE`, the `minDepthBounds` and `maxDepthBounds` members of `pDepthStencilState` **must** be between `0.0` and `1.0`, inclusive
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`, and the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure chained to the `pNext` chain of `pMultisampleState` is `VK_TRUE`, `sampleLocationsInfo.sampleLocationGridSize.width` **must** evenly divide `VkMultisamplePropertiesEXT::sampleLocationGridSize.width` as returned by `vkGetPhysicalDeviceMultisamplePropertiesEXT` with a `samples` parameter equaling `rasterizationSamples`
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`, and the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure chained to the `pNext` chain of `pMultisampleState` is `VK_TRUE`, `sampleLocationsInfo.sampleLocationGridSize.height` **must** evenly divide `VkMultisamplePropertiesEXT::sampleLocationGridSize.height` as returned by `vkGetPhysicalDeviceMultisamplePropertiesEXT` with a `samples` parameter equaling `rasterizationSamples`
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`, and the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure chained to the `pNext` chain of `pMultisampleState` is `VK_TRUE`, `sampleLocationsInfo.sampleLocationsPerPixel` **must** equal `rasterizationSamples`
- If the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure chained to the `pNext` chain of `pMultisampleState` is `VK_TRUE`, the fragment shader code **must** not statically use the extended instruction `InterpolateAtSample`
- `layout` **must** be consistent with all shaders specified in `pStages`
- If neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, and if `subpass` uses color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` **must** be the same as the sample

count for those subpass attachments

- If the `VK_AMD_mixed_attachment_samples` extension is enabled, and if `subpass` uses color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` **must** equal the maximum of the sample counts of those subpass attachments
- If the `VK_NV_framebuffer_mixed_samples` extension is enabled, and if `subpass` has a depth/stencil attachment and depth test, stencil test, or depth bounds test are enabled, then the `rasterizationSamples` member of `pMultisampleState` **must** be the same as the sample count of the depth/stencil attachment
- If the `VK_NV_framebuffer_mixed_samples` extension is enabled, and if `subpass` has any color attachments, then the `rasterizationSamples` member of `pMultisampleState` **must** be greater than or equal to the sample count for those subpass attachments
- If the `VK_NV_coverage_reduction_mode` extension is enabled, the coverage reduction mode specified by `VkPipelineCoverageReductionStateCreateInfoNV::coverageReductionMode`, the `rasterizationSamples` member of `pMultisampleState` and the sample counts for the color and depth/stencil attachments (if the subpass has them) **must** be a valid combination returned by `vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV`
- If `subpass` does not use any color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` **must** follow the rules for a `zero-attachment` subpass
- `subpass` **must** be a valid subpass within `renderPass`
- If the `renderPass` has multiview enabled and `subpass` has more than one bit set in the view mask and `multiviewTessellationShader` is not enabled, then `pStages` **must** not include tessellation shaders.
- If the `renderPass` has multiview enabled and `subpass` has more than one bit set in the view mask and `multiviewGeometryShader` is not enabled, then `pStages` **must** not include a geometry shader.
- If the `renderPass` has multiview enabled and `subpass` has more than one bit set in the view mask, shaders in the pipeline **must** not write to the `Layer` built-in output
- If the `renderPass` has multiview enabled, then all shaders **must** not include variables decorated with the `Layer` built-in decoration in their interfaces.
- `flags` **must** not contain the `VK_PIPELINE_CREATE_DISPATCH_BASE` flag.
- If `pStages` includes a fragment shader stage and an input attachment was referenced by the `VkRenderPassInputAttachmentAspectCreateInfo` at `renderPass` create time, its shader code **must** not read from any aspect that was not specified in the `aspectMask` of the corresponding `VkInputAttachmentAspectReference` structure.
- The number of resources in `layout` accessible to each shader stage that is used by the pipeline **must** be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV`, and the `viewportWScaleEnable` member of a `VkPipelineViewportWScaleCreateInfoNV` structure, chained to the `pNext` chain of `pViewportState`, is `VK_TRUE`, the `pViewportWScalings` member of the

`VkPipelineViewportWScalingStateCreateInfoNV` **must** be a pointer to an array of `VkPipelineViewportWScalingStateCreateInfoNV::viewportCount` valid `VkViewportWScalingNV` structures

- If `pStages` includes a vertex shader stage, `pVertexInputState` **must** be a valid pointer to a valid `VkPipelineVertexInputStateCreateInfo` structure
- If `pStages` includes a vertex shader stage, `pInputAssemblyState` **must** be a valid pointer to a valid `VkPipelineInputAssemblyStateCreateInfo` structure
- The `Xfb` execution mode **can** be specified by only one shader stage in `pStages`
- If any shader stage in `pStages` specifies `Xfb` execution mode it **must** be the last vertex processing stage
- If a `VkPipelineRasterizationStateStreamCreateInfoEXT::rasterizationStream` value other than zero is specified, all variables in the output interface of the entry point being compiled decorated with `Position`, `PointSize`, `ClipDistance`, or `CullDistance` **must** all be decorated with identical `Stream` values that match the `rasterizationStream`
- If `VkPipelineRasterizationStateStreamCreateInfoEXT::rasterizationStream` is zero, or not specified, all variables in the output interface of the entry point being compiled decorated with `Position`, `PointSize`, `ClipDistance`, or `CullDistance` **must** all be decorated with a `Stream` value of zero, or **must** not specify the `Stream` decoration
- If the last vertex processing stage is a geometry shader, and that geometry shader uses the `GeometryStreams` capability, then `VkPhysicalDeviceTransformFeedbackFeaturesEXT::geometryStreams` feature **must** be enabled
- If there are any mesh shader stages in the pipeline there **must** not be any shader stage in the pipeline with a `Xfb` execution mode.
- If the `lineRasterizationMode` member of a `VkPipelineRasterizationLineStateCreateInfoEXT` structure chained to the `pNext` chain of `pRasterizationState` is `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` or `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` and if rasterization is enabled, then the `alphaToCoverageEnable`, `alphaToOneEnable`, and `sampleShadingEnable` members of `pMultisampleState` **must** all be `VK_FALSE`
- If the `stippledLineEnable` member of `VkPipelineRasterizationLineStateCreateInfoEXT` is `VK_TRUE` and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_LINE_STIPPLE_EXT`, then the `lineStippleFactor` member of `VkPipelineRasterizationLineStateCreateInfoEXT` **must** be in the range [1,256]

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkPipelineCompilerControlCreateInfoAMD`, `VkPipelineCreationFeedbackCreateInfoEXT`, `VkPipelineDiscardRectangleStateCreateInfoEXT`, `VkPipelineRepresentativeFragmentTestStateCreateInfoNV` or `VkPipelineRepresentativeFragmentTestStateCreateInfoNV`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be a valid combination of `VkPipelineCreateFlagBits` values
- `pStages` **must** be a valid pointer to an array of `stageCount` valid `VkPipelineShaderStageCreateInfo` structures
- `pRasterizationState` **must** be a valid pointer to a valid `VkPipelineRasterizationStateCreateInfo` structure
- If `pDynamicState` is not `NULL`, `pDynamicState` **must** be a valid pointer to a valid `VkPipelineDynamicStateCreateInfo` structure
- `layout` **must** be a valid `VkPipelineLayout` handle
- `renderPass` **must** be a valid `VkRenderPass` handle
- `stageCount` **must** be greater than `0`
- Each of `basePipelineHandle`, `layout`, and `renderPass` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

Possible values of the `flags` member of `VkGraphicsPipelineCreateInfo`, `VkComputePipelineCreateInfo`, and `VkRayTracingPipelineCreateInfoNV`, specifying how a pipeline is created, are:

```
typedef enum VkPipelineCreateFlagBits {
    VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT = 0x00000001,
    VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT = 0x00000002,
    VK_PIPELINE_CREATE_DERIVATIVE_BIT = 0x00000004,
    VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT = 0x00000008,
    VK_PIPELINE_CREATE_DISPATCH_BASE_BIT = 0x00000010,
    VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV = 0x00000020,
    VK_PIPELINE_CREATE_CAPTURE_STATISTICS_BIT_KHR = 0x00000040,
    VK_PIPELINE_CREATE_CAPTURE_INTERNAL REPRESENTATIONS_BIT_KHR = 0x00000080,
    VK_PIPELINE_CREATE_DISPATCH_BASE = VK_PIPELINE_CREATE_DISPATCH_BASE_BIT,
    VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT_KHR =
VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT,
    VK_PIPELINE_CREATE_DISPATCH_BASE_KHR = VK_PIPELINE_CREATE_DISPATCH_BASE,
    VK_PIPELINE_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPipelineCreateFlagBits;
```

- `VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT` specifies that the created pipeline will not be

optimized. Using this flag **may** reduce the time taken to create the pipeline.

- **VK\_PIPELINE\_CREATE\_ALLOW\_DERIVATIVES\_BIT** specifies that the pipeline to be created is allowed to be the parent of a pipeline that will be created in a subsequent call to [vkCreateGraphicsPipelines](#) or [vkCreateComputePipelines](#).
- **VK\_PIPELINE\_CREATE\_DERIVATIVE\_BIT** specifies that the pipeline to be created will be a child of a previously created parent pipeline.
- **VK\_PIPELINE\_CREATE\_VIEW\_INDEX\_FROM\_DEVICE\_INDEX\_BIT** specifies that any shader input variables decorated as *ViewIndex* will be assigned values as if they were decorated as *DeviceIndex*.
- **VK\_PIPELINE\_CREATE\_DISPATCH\_BASE** specifies that a compute pipeline **can** be used with [vkCmdDispatchBase](#) with a non-zero base workgroup.
- **VK\_PIPELINE\_CREATE\_DEFER\_COMPILE\_BIT\_NV** specifies that a pipeline is created with all shaders in the deferred state. Before using the pipeline the application **must** call [vkCompileDeferredNV](#) exactly once on each shader in the pipeline before using the pipeline.
- **VK\_PIPELINE\_CREATE\_CAPTURE\_STATISTICS\_BIT\_KHR** specifies that the shader compiler should capture statistics for the executables produced by the compile process which **can** later be retrieved by calling [vkGetPipelineExecutableStatisticsKHR](#). Enabling this flag **must** not affect the final compiled pipeline but **may** disable pipeline caching or otherwise affect pipeline creation time.
- **VK\_PIPELINE\_CREATE\_CAPTURE\_INTERNAL REPRESENTATIONS\_BIT\_KHR** specifies that the shader compiler should capture the internal representations of executables produced by the compile process which **can** later be retrieved by calling [vkGetPipelineExecutableInternalRepresentationsKHR](#). Enabling this flag **must** not affect the final compiled pipeline but **may** disable pipeline caching or otherwise affect pipeline creation time.

It is valid to set both **VK\_PIPELINE\_CREATE\_ALLOW\_DERIVATIVES\_BIT** and **VK\_PIPELINE\_CREATE\_DERIVATIVE\_BIT**. This allows a pipeline to be both a parent and possibly a child in a pipeline hierarchy. See [Pipeline Derivatives](#) for more information.

```
typedef VkFlags VkPipelineCreateFlags;
```

`VkPipelineCreateFlags` is a bitmask type for setting a mask of zero or more [VkPipelineCreateFlagBits](#).

The `VkPipelineDynamicStateCreateInfo` structure is defined as:

```
typedef struct VkPipelineDynamicStateCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineDynamicStateCreateFlags flags;
    uint32_t                  dynamicStateCount;
    const VkDynamicState*     pDynamicStates;
} VkPipelineDynamicStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `dynamicStateCount` is the number of elements in the `pDynamicStates` array.
- `pDynamicStates` is a pointer to an array of `VkDynamicState` values specifying which pieces of pipeline state will use the values from dynamic state commands rather than from pipeline state creation info.

### Valid Usage

- Each element of `pDynamicStates` **must** be unique

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- If `dynamicStateCount` is not `0`, `pDynamicStates` **must** be a valid pointer to an array of `dynamicStateCount` valid `VkDynamicState` values

```
typedef VkFlags VkPipelineDynamicStateCreateInfo;
```

`VkPipelineDynamicStateCreateInfo` is a bitmask type for setting a mask, but is currently reserved for future use.

The source of different pieces of dynamic state is specified by the `VkPipelineDynamicStateCreateInfo::pDynamicStates` property of the currently active pipeline, each of whose elements **must** be one of the values:

```

typedef enum VkDynamicState {
    VK_DYNAMIC_STATE_VIEWPORT = 0,
    VK_DYNAMIC_STATE_SCISSOR = 1,
    VK_DYNAMIC_STATE_LINE_WIDTH = 2,
    VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
    VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
    VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
    VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK = 6,
    VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
    VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
    VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV = 1000087000,
    VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT = 1000099000,
    VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT = 1000143000,
    VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV = 1000164004,
    VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV = 1000164006,
    VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV = 1000205001,
    VK_DYNAMIC_STATE_LINE_STIPPLE_EXT = 1000259000,
    VK_DYNAMIC_STATE_MAX_ENUM = 0x7FFFFFFF
} VkDynamicState;

```

- `VK_DYNAMIC_STATE_VIEWPORT` specifies that the `pViewports` state in `VkPipelineViewportStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetViewport` before any draw commands. The number of viewports used by a pipeline is still specified by the `viewportCount` member of `VkPipelineViewportStateCreateInfo`.
- `VK_DYNAMIC_STATE_SCISSOR` specifies that the `pScissors` state in `VkPipelineViewportStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetScissor` before any draw commands. The number of scissor rectangles used by a pipeline is still specified by the `scissorCount` member of `VkPipelineViewportStateCreateInfo`.
- `VK_DYNAMIC_STATE_LINE_WIDTH` specifies that the `lineWidth` state in `VkPipelineRasterizationStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetLineWidth` before any draw commands that generate line primitives for the rasterizer.
- `VK_DYNAMIC_STATE_DEPTH_BIAS` specifies that the `depthBiasConstantFactor`, `depthBiasClamp` and `depthBiasSlopeFactor` states in `VkPipelineRasterizationStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetDepthBias` before any draws are performed with `depthBiasEnable` in `VkPipelineRasterizationStateCreateInfo` set to `VK_TRUE`.
- `VK_DYNAMIC_STATE_BLEND_CONSTANTS` specifies that the `blendConstants` state in `VkPipelineColorBlendStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetBlendConstants` before any draws are performed with a pipeline state with `VkPipelineColorBlendAttachmentState` member `blendEnable` set to `VK_TRUE` and any of the blend functions using a constant blend color.
- `VK_DYNAMIC_STATE_DEPTH_BOUNDS` specifies that the `minDepthBounds` and `maxDepthBounds` states of `VkPipelineDepthStencilStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetDepthBounds` before any draws are performed with a pipeline state with `VkPipelineDepthStencilStateCreateInfo` member `depthBoundsTestEnable` set to `VK_TRUE`.

- **VK\_DYNAMIC\_STATE\_STENCIL\_COMPARE\_MASK** specifies that the `compareMask` state in `VkPipelineDepthStencilStateCreateInfo` for both `front` and `back` will be ignored and **must** be set dynamically with `vkCmdSetStencilCompareMask` before any draws are performed with a pipeline state with `VkPipelineDepthStencilStateCreateInfo` member `stencilTestEnable` set to `VK_TRUE`
- **VK\_DYNAMIC\_STATE\_STENCIL\_WRITE\_MASK** specifies that the `writeMask` state in `VkPipelineDepthStencilStateCreateInfo` for both `front` and `back` will be ignored and **must** be set dynamically with `vkCmdSetStencilWriteMask` before any draws are performed with a pipeline state with `VkPipelineDepthStencilStateCreateInfo` member `stencilTestEnable` set to `VK_TRUE`
- **VK\_DYNAMIC\_STATE\_STENCIL\_REFERENCE** specifies that the `reference` state in `VkPipelineDepthStencilStateCreateInfo` for both `front` and `back` will be ignored and **must** be set dynamically with `vkCmdSetStencilReference` before any draws are performed with a pipeline state with `VkPipelineDepthStencilStateCreateInfo` member `stencilTestEnable` set to `VK_TRUE`
- **VK\_DYNAMIC\_STATE\_VIEWPORT\_W\_SCALING\_NV** specifies that the `pViewportScalings` state in `VkPipelineViewportWScalingStateCreateInfoNV` will be ignored and **must** be set dynamically with `vkCmdSetViewportWScalingNV` before any draws are performed with a pipeline state with `VkPipelineViewportWScalingStateCreateInfoNV` member `viewportScalingEnable` set to `VK_TRUE`
- **VK\_DYNAMIC\_STATE\_DISCARD\_RECTANGLE\_EXT** specifies that the `pDiscardRectangles` state in `VkPipelineDiscardRectangleStateCreateInfoEXT` will be ignored and **must** be set dynamically with `vkCmdSetDiscardRectangleEXT` before any draw or clear commands. The `VkDiscardRectangleModeEXT` and the number of active discard rectangles is still specified by the `discardRectangleMode` and `discardRectangleCount` members of `VkPipelineDiscardRectangleStateCreateInfoEXT`.
- **VK\_DYNAMIC\_STATE\_SAMPLE\_LOCATIONS\_EXT** specifies that the `sampleLocationsInfo` state in `VkPipelineSampleLocationsStateCreateInfoEXT` will be ignored and **must** be set dynamically with `vkCmdSetSampleLocationsEXT` before any draw or clear commands. Enabling custom sample locations is still indicated by the `sampleLocationsEnable` member of `VkPipelineSampleLocationsStateCreateInfoEXT`.
- **VK\_DYNAMIC\_STATE\_EXCLUSIVE\_SCISSOR\_NV** specifies that the `pExclusiveScissors` state in `VkPipelineViewportExclusiveScissorStateCreateInfoNV` will be ignored and **must** be set dynamically with `vkCmdSetExclusiveScissorNV` before any draw commands. The number of exclusive scissor rectangles used by a pipeline is still specified by the `exclusiveScissorCount` member of `VkPipelineViewportExclusiveScissorStateCreateInfoNV`.
- **VK\_DYNAMIC\_STATE\_VIEWPORT\_SHADING\_RATE\_PALETTE\_NV** specifies that the `pShadingRatePalettes` state in `VkPipelineViewportShadingRateImageStateCreateInfoNV` will be ignored and **must** be set dynamically with `vkCmdSetViewportShadingRatePaletteNV` before any draw commands.
- **VK\_DYNAMIC\_STATE\_VIEWPORT\_COARSE\_SAMPLE\_ORDER\_NV** specifies that the coarse sample order state in `VkPipelineViewportCoarseSampleOrderStateCreateInfoNV` will be ignored and **must** be set dynamically with `vkCmdSetCoarseSampleOrderNV` before any draw commands.
- **VK\_DYNAMIC\_STATE\_LINE\_STIPPLE\_EXT** specifies that the `lineStippleFactor` and `lineStipplePattern` state in `VkPipelineRasterizationLineStateCreateInfoEXT` will be ignored and **must** be set dynamically with `vkCmdSetLineStippleEXT` before any draws are performed with a pipeline state with `VkPipelineRasterizationLineStateCreateInfoEXT` member `stippledLineEnable` set to `VK_TRUE`.

### 9.2.1. Valid Combinations of Stages for Graphics Pipelines

The geometric primitive processing can either be handled on a per primitive basis by the vertex, tessellation, and geometry shader stages, or on a per mesh basis using task and mesh shader stages. If the pipeline includes a mesh shader stage, it uses the mesh pipeline, otherwise it uses the primitive pipeline.

If a task shader is omitted, the task shading stage is skipped.

If tessellation shader stages are omitted, the tessellation shading and fixed-function stages of the pipeline are skipped.

If a geometry shader is omitted, the geometry shading stage is skipped.

If a fragment shader is omitted, fragment color outputs have undefined values, and the fragment depth value is unmodified. This **can** be useful for depth-only rendering.

Presence of a shader stage in a pipeline is indicated by including a valid [VkPipelineShaderCreateInfo](#) with `module` and `pName` selecting an entry point from a shader module, where that entry point is valid for the stage specified by `stage`.

Presence of some of the fixed-function stages in the pipeline is implicitly derived from enabled shaders and provided state. For example, the fixed-function tessellator is always present when the pipeline has valid Tessellation Control and Tessellation Evaluation shaders.

*For example:*

- Depth/stencil-only rendering in a subpass with no color attachments
  - Active Pipeline Shader Stages
    - Vertex Shader
  - Required: Fixed-Function Pipeline Stages
    - [VkPipelineVertexInputStateCreateInfo](#)
    - [VkPipelineAssemblyStateCreateInfo](#)
    - [VkPipelineViewportStateCreateInfo](#)
    - [VkPipelineRasterizationStateCreateInfo](#)
    - [VkPipelineMultisampleStateCreateInfo](#)
    - [VkPipelineDepthStencilStateCreateInfo](#)
- Color-only rendering in a subpass with no depth/stencil attachment
  - Active Pipeline Shader Stages
    - Vertex Shader
    - Fragment Shader
  - Required: Fixed-Function Pipeline Stages
    - [VkPipelineVertexInputStateCreateInfo](#)
    - [VkPipelineAssemblyStateCreateInfo](#)

- [VkPipelineViewportStateCreateInfo](#)
- [VkPipelineRasterizationStateCreateInfo](#)
- [VkPipelineMultisampleStateCreateInfo](#)
- [VkPipelineColorBlendStateCreateInfo](#)
- Rendering pipeline with tessellation and geometry shaders
  - Active Pipeline Shader Stages
    - Vertex Shader
    - Tessellation Control Shader
    - Tessellation Evaluation Shader
    - Geometry Shader
    - Fragment Shader
  - Required: Fixed-Function Pipeline Stages
    - [VkPipelineVertexInputStateCreateInfo](#)
    - [VkPipelineInputAssemblyStateCreateInfo](#)
    - [VkPipelineTessellationStateCreateInfo](#)
    - [VkPipelineViewportStateCreateInfo](#)
    - [VkPipelineRasterizationStateCreateInfo](#)
    - [VkPipelineMultisampleStateCreateInfo](#)
    - [VkPipelineDepthStencilStateCreateInfo](#)
    - [VkPipelineColorBlendStateCreateInfo](#)
- Rendering pipeline with task and mesh shaders
  - Active Pipeline Shader Stages
    - Task Shader
    - Mesh Shader
    - Fragment Shader
  - Required: Fixed-Function Pipeline Stages
    - [VkPipelineViewportStateCreateInfo](#)
    - [VkPipelineRasterizationStateCreateInfo](#)
    - [VkPipelineMultisampleStateCreateInfo](#)
    - [VkPipelineDepthStencilStateCreateInfo](#)
    - [VkPipelineColorBlendStateCreateInfo](#)

## 9.3. Pipeline destruction

To destroy a graphics or compute pipeline, call:

```
void vkDestroyPipeline(  
    VkDevice device,  
    VkPipeline pipeline,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the pipeline.
- `pipeline` is the handle of the pipeline to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

### Valid Usage

- All submitted commands that refer to `pipeline` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `pipeline` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `pipeline` was created, `pAllocator` **must** be `NULL`

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `pipeline` is not `VK_NULL_HANDLE`, `pipeline` **must** be a valid `VkPipeline` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `pipeline` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `pipeline` **must** be externally synchronized

## 9.4. Multiple Pipeline Creation

Multiple pipelines **can** be created simultaneously by passing an array of `VkGraphicsPipelineCreateInfo` or `VkComputePipelineCreateInfo` structures into the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` commands, respectively. Applications **can** group together similar pipelines to be created in a single call, and implementations are encouraged to look for reuse opportunities within a group-create.

When an application attempts to create many pipelines in a single command, it is possible that some subset **may** fail creation. In that case, the corresponding entries in the `pPipelines` output array will be filled with `VK_NULL_HANDLE` values. If any pipeline fails creation (for example, due to out of memory errors), the `vkCreate*Pipelines` commands will return an error code. The

implementation will attempt to create all pipelines, and only return `VK_NULL_HANDLE` values for those that actually failed.

## 9.5. Pipeline Derivatives

A pipeline derivative is a child pipeline created from a parent pipeline, where the child and parent are expected to have much commonality. The goal of derivative pipelines is that they be cheaper to create using the parent as a starting point, and that it be more efficient (on either host or device) to switch/bind between children of the same parent.

A derivative pipeline is created by setting the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag in the `VkPipelineCreateInfo` structure. If this is set, then exactly one of `basePipelineHandle` or `basePipelineIndex` members of the structure **must** have a valid handle/index, and specifies the parent pipeline. If `basePipelineHandle` is used, the parent pipeline **must** have already been created. If `basePipelineIndex` is used, then the parent is being created in the same command. `VK_NULL_HANDLE` acts as the invalid handle for `basePipelineHandle`, and -1 is the invalid index for `basePipelineIndex`. If `basePipelineIndex` is used, the base pipeline **must** appear earlier in the array. The base pipeline **must** have been created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` flag set.

## 9.6. Pipeline Cache

Pipeline cache objects allow the result of pipeline construction to be reused between pipelines and between runs of an application. Reuse between pipelines is achieved by passing the same pipeline cache object when creating multiple related pipelines. Reuse across runs of an application is achieved by retrieving pipeline cache contents in one run of an application, saving the contents, and using them to preinitialize a pipeline cache on a subsequent run. The contents of the pipeline cache objects are managed by the implementation. Applications **can** manage the host memory consumed by a pipeline cache object and control the amount of data retrieved from a pipeline cache object.

Pipeline cache objects are represented by `VkPipelineCache` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineCache)
```

To create pipeline cache objects, call:

```
VkResult vkCreatePipelineCache(  
    VkDevice                                     device,  
    const VkPipelineCacheCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks*     pAllocator,  
    VkPipelineCache*                pPipelineCache);
```

- `device` is the logical device that creates the pipeline cache object.
- `pCreateInfo` is a pointer to a `VkPipelineCacheCreateInfo` structure containing initial parameters for the pipeline cache object.

- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pPipelineCache` is a pointer to a `VkPipelineCache` handle in which the resulting pipeline cache object is returned.

*Note*



Applications **can** track and manage the total host memory size of a pipeline cache object using the `pAllocator`. Applications **can** limit the amount of data retrieved from a pipeline cache object in `vkGetPipelineCacheData`. Implementations **should** not internally limit the total number of entries added to a pipeline cache object or the total host memory consumed.

Once created, a pipeline cache **can** be passed to the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` commands. If the pipeline cache passed into these commands is not `VK_NULL_HANDLE`, the implementation will query it for possible reuse opportunities and update it with new content. The use of the pipeline cache object in these commands is internally synchronized, and the same pipeline cache object **can** be used in multiple threads simultaneously.

*Note*



Implementations **should** make every effort to limit any critical sections to the actual accesses to the cache, which is expected to be significantly shorter than the duration of the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` commands.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkPipelineCacheCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pPipelineCache` **must** be a valid pointer to a `VkPipelineCache` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkPipelineCacheCreateInfo` structure is defined as:

```
typedef struct VkPipelineCacheCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkPipelineCacheCreateFlags flags;
    size_t                     initialDataSize;
    const void*                pInitialData;
} VkPipelineCacheCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `initialDataSize` is the number of bytes in `pInitialData`. If `initialDataSize` is zero, the pipeline cache will initially be empty.
- `pInitialData` is a pointer to previously retrieved pipeline cache data. If the pipeline cache data is incompatible (as defined below) with the device, the pipeline cache will be initially empty. If `initialDataSize` is zero, `pInitialData` is ignored.

## Valid Usage

- If `initialDataSize` is not `0`, it **must** be equal to the size of `pInitialData`, as returned by `vkGetPipelineCacheData` when `pInitialData` was originally retrieved
- If `initialDataSize` is not `0`, `pInitialData` **must** have been retrieved from a previous call to `vkGetPipelineCacheData`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- If `initialDataSize` is not `0`, `pInitialData` **must** be a valid pointer to an array of `initialDataSize` bytes

```
typedef VkFlags VkPipelineCacheCreateFlags;
```

`VkPipelineCacheCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

Pipeline cache objects **can** be merged using the command:

```
VkResult vkMergePipelineCaches(  
    VkDevice device,  
    VkPipelineCache dstCache,  
    uint32_t srcCacheCount,  
    const VkPipelineCache* pSrcCaches);
```

- `device` is the logical device that owns the pipeline cache objects.
- `dstCache` is the handle of the pipeline cache to merge results into.
- `srcCacheCount` is the length of the `pSrcCaches` array.
- `pSrcCaches` is a pointer to an array of pipeline cache handles, which will be merged into `dstCache`. The previous contents of `dstCache` are included after the merge.

*Note*



The details of the merge operation are implementation dependent, but implementations **should** merge the contents of the specified pipelines and prune duplicate entries.

## Valid Usage

- `dstCache` **must** not appear in the list of source caches

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `dstCache` **must** be a valid `VkPipelineCache` handle
- `pSrcCaches` **must** be a valid pointer to an array of `srcCacheCount` valid `VkPipelineCache` handles
- `srcCacheCount` **must** be greater than `0`
- `dstCache` **must** have been created, allocated, or retrieved from `device`
- Each element of `pSrcCaches` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `dstCache` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Data **can** be retrieved from a pipeline cache object using the command:

```
VkResult vkGetPipelineCacheData(  
    VkDevice device,  
    VkPipelineCache pipelineCache,  
    size_t* pDatasize,  
    void* pData);
```

- `device` is the logical device that owns the pipeline cache.
- `pipelineCache` is the pipeline cache to retrieve data from.
- `pDatasize` is a pointer to a `size_t` value related to the amount of data in the pipeline cache, as described below.
- `pData` is either `NULL` or a pointer to a buffer.

If `pData` is `NULL`, then the maximum size of the data that **can** be retrieved from the pipeline cache, in bytes, is returned in `pDatasize`. Otherwise, `pDatasize` **must** point to a variable set by the user to the size of the buffer, in bytes, pointed to by `pData`, and on return the variable is overwritten with the amount of data actually written to `pData`.

If `pDatasize` is less than the maximum size that **can** be retrieved by the pipeline cache, at most `pDatasize` bytes will be written to `pData`, and `vkGetPipelineCacheData` will return `VK_INCOMPLETE`. Any data written to `pData` is valid and **can** be provided as the `pInitialData` member of the `VkPipelineCacheCreateInfo` structure passed to `vkCreatePipelineCache`.

Two calls to `vkGetPipelineCacheData` with the same parameters **must** retrieve the same data unless a command that modifies the contents of the cache is called between them.

Applications **can** store the data retrieved from the pipeline cache, and use these data, possibly in a future run of the application, to populate new pipeline cache objects. The results of pipeline compiles, however, **may** depend on the vendor ID, device ID, driver version, and other details of the device. To enable applications to detect when previously retrieved data is incompatible with the device, the initial bytes written to `pData` **must** be a header consisting of the following members:

*Table 12. Layout for pipeline cache header version `VK_PIPELINE_CACHE_HEADER_VERSION_ONE`*

Offset	Size	Meaning
0	4	length in bytes of the entire pipeline cache header written as a stream of bytes, with the least significant byte first
4	4	a <code>VkPipelineCacheHeaderVersion</code> value written as a stream of bytes, with the least significant byte first
8	4	a vendor ID equal to <code>VkPhysicalDeviceProperties::vendorID</code> written as a stream of bytes, with the least significant byte first
12	4	a device ID equal to <code>VkPhysicalDeviceProperties::deviceID</code> written as a stream of bytes, with the least significant byte first
16	<code>VK_UUID_SIZE</code>	a pipeline cache ID equal to <code>VkPhysicalDeviceProperties::pipelineCacheUUID</code>

The first four bytes encode the length of the entire pipeline cache header, in bytes. This value includes all fields in the header including the pipeline cache version field and the size of the length field.

The next four bytes encode the pipeline cache version, as described for `VkPipelineCacheHeaderVersion`. A consumer of the pipeline cache **should** use the cache version to interpret the remainder of the cache header.

If `pDataSize` is less than what is necessary to store this header, nothing will be written to `pData` and zero will be written to `pDataSize`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pipelineCache` **must** be a valid `VkPipelineCache` handle
- `pDataSize` **must** be a valid pointer to a `size_t` value
- If the value referenced by `pDataSize` is not `0`, and `pData` is not `NULL`, `pData` **must** be a valid pointer to an array of `pDataSize` bytes
- `pipelineCache` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Possible values of the second group of four bytes in the header returned by `vkGetPipelineCacheData`, encoding the pipeline cache version, are:

```
typedef enum VkPipelineCacheHeaderVersion {
    VK_PIPELINE_CACHE_HEADER_VERSION_ONE = 1,
    VK_PIPELINE_CACHE_HEADER_VERSION_MAX_ENUM = 0x7FFFFFFF
} VkPipelineCacheHeaderVersion;
```

- `VK_PIPELINE_CACHE_HEADER_VERSION_ONE` specifies version one of the pipeline cache.

To destroy a pipeline cache, call:

```
void vkDestroyPipelineCache(
    VkDevice                                     device,
    VkPipelineCache                             pipelineCache,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the pipeline cache object.
- `pipelineCache` is the handle of the pipeline cache to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- If `VkAllocationCallbacks` were provided when `pipelineCache` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `pipelineCache` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `pipelineCache` is not `VK_NULL_HANDLE`, `pipelineCache` **must** be a valid `VkPipelineCache` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `pipelineCache` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `pipelineCache` **must** be externally synchronized

## 9.7. Specialization Constants

Specialization constants are a mechanism whereby constants in a SPIR-V module **can** have their constant value specified at the time the `VkPipeline` is created. This allows a SPIR-V module to have constants that **can** be modified while executing an application that uses the Vulkan API.

*Note*



Specialization constants are useful to allow a compute shader to have its local workgroup size changed at runtime by the user, for example.

Each instance of the `VkPipelineShaderStageCreateInfo` structure contains a parameter `pSpecializationInfo`, which **can** be `NULL` to indicate no specialization constants, or point to a `VkSpecializationInfo` structure.

The `VkSpecializationInfo` structure is defined as:

```
typedef struct VkSpecializationInfo {
    uint32_t                         mapEntryCount;
    const VkSpecializationMapEntry*   pMapEntries;
    size_t                            dataSize;
    const void*                       pData;
} VkSpecializationInfo;
```

- `mapEntryCount` is the number of entries in the `pMapEntries` array.
- `pMapEntries` is a pointer to an array of `VkSpecializationMapEntry` structures which map constant IDs to offsets in `pData`.
- `dataSize` is the byte size of the `pData` buffer.
- `pData` contains the actual constant values to specialize with.

`pMapEntries` is a pointer to a `VkSpecializationMapEntry` structure.

### Valid Usage

- The `offset` member of each element of `pMapEntries` **must** be less than `dataSize`
- The `size` member of each element of `pMapEntries` **must** be less than or equal to `dataSize` minus `offset`

### Valid Usage (Implicit)

- If `mapEntryCount` is not `0`, `pMapEntries` **must** be a valid pointer to an array of `mapEntryCount` valid `VkSpecializationMapEntry` structures
- If `dataSize` is not `0`, `pData` **must** be a valid pointer to an array of `dataSize` bytes

The `VkSpecializationMapEntry` structure is defined as:

```
typedef struct VkSpecializationMapEntry {
    uint32_t constantID;
    uint32_t offset;
    size_t size;
} VkSpecializationMapEntry;
```

- `constantID` is the ID of the specialization constant in SPIR-V.
- `offset` is the byte offset of the specialization constant value within the supplied data buffer.
- `size` is the byte size of the specialization constant value within the supplied data buffer.

If a `constantID` value is not a specialization constant ID used in the shader, that map entry does not affect the behavior of the pipeline.

## Valid Usage

- For a `constantID` specialization constant declared in a shader, `size` **must** match the byte size of the `constantID`. If the specialization constant is of type `boolean`, `size` **must** be the byte size of `VkBool32`

In human readable SPIR-V:

```
OpDecorate %x SpecId 13 ; decorate .x component of WorkgroupSize with ID 13
OpDecorate %y SpecId 42 ; decorate .y component of WorkgroupSize with ID 42
OpDecorate %z SpecId 3 ; decorate .z component of WorkgroupSize with ID 3
OpDecorate %wgsiz3 BuiltIn WorkgroupSize ; decorate WorkgroupSize onto constant
%i32 = OpTypeInt 32 0 ; declare an unsigned 32-bit type
%uvec3 = OpTypeVector %i32 3 ; declare a 3 element vector type of unsigned 32-bit
%x = OpSpecConstant %i32 1 ; declare the .x component of WorkgroupSize
%y = OpSpecConstant %i32 1 ; declare the .y component of WorkgroupSize
%z = OpSpecConstant %i32 1 ; declare the .z component of WorkgroupSize
%wgsiz3 = OpSpecConstantComposite %uvec3 %x %y %z ; declare WorkgroupSize
```

From the above we have three specialization constants, one for each of the x, y & z elements of the WorkgroupSize vector.

Now to specialize the above via the specialization constants mechanism:

```

const VkSpecializationMapEntry entries[] =
{
{
    13,                                // constantID
    0 * sizeof(uint32_t),           // offset
    sizeof(uint32_t)                  // size
},
{
    42,                                // constantID
    1 * sizeof(uint32_t),           // offset
    sizeof(uint32_t)                  // size
},
{
    3,                                // constantID
    2 * sizeof(uint32_t),           // offset
    sizeof(uint32_t)                  // size
}
};

const uint32_t data[] = { 16, 8, 4 }; // our workgroup size is 16x8x4

const VkSpecializationInfo info =
{
    3,                                // mapEntryCount
    entries,                            // pMapEntries
    3 * sizeof(uint32_t),           // dataSize
    data,                               // pData
};

```

Then when calling `vkCreateComputePipelines`, and passing the `VkSpecializationInfo` we defined as the `pSpecializationInfo` parameter of `VkPipelineShaderStageCreateInfo`, we will create a compute pipeline with the runtime specified local workgroup size.

Another example would be that an application has a SPIR-V module that has some platform-dependent constants they wish to use.

In human readable SPIR-V:

```

OpDecorate %1 SpecId 0 ; decorate our signed 32-bit integer constant
OpDecorate %2 SpecId 12 ; decorate our 32-bit floating-point constant
%i32 = OpTypeInt 32 1 ; declare a signed 32-bit type
%float = OpTypeFloat 32 ; declare a 32-bit floating-point type
%1 = OpSpecConstant %i32 -1 ; some signed 32-bit integer constant
%2 = OpSpecConstant %float 0.5 ; some 32-bit floating-point constant

```

From the above we have two specialization constants, one is a signed 32-bit integer and the second is a 32-bit floating-point.

Now to specialize the above via the specialization constants mechanism:

```
struct SpecializationData {
    int32_t data0;
    float data1;
};

const VkSpecializationMapEntry entries[] =
{
{
    0,                                // constantID
    offsetof(SpecializationData, data0), // offset
    sizeof(SpecializationData::data0)   // size
},
{
    12,                               // constantID
    offsetof(SpecializationData, data1), // offset
    sizeof(SpecializationData::data1)   // size
}
};

SpecializationData data;
data.data0 = -42;      // set the data for the 32-bit integer
data.data1 = 42.0f;    // set the data for the 32-bit floating-point

const VkSpecializationInfo info =
{
    2,                                // mapEntryCount
    entries,                           // pMapEntries
    sizeof(data),                      // dataSize
    &data,                            // pData
};
```

It is legal for a SPIR-V module with specializations to be compiled into a pipeline where no specialization info was provided. SPIR-V specialization constants contain default values such that if a specialization is not provided, the default value will be used. In the examples above, it would be valid for an application to only specialize some of the specialization constants within the SPIR-V module, and let the other constants use their default values encoded within the OpSpecConstant declarations.

## 9.8. Pipeline Binding

Once a pipeline has been created, it **can** be bound to the command buffer using the command:

```
void vkCmdBindPipeline(  
    VkCommandBuffer  
    pipelineBindPoint  
    VkPipeline  
    pipeline);
```

- `commandBuffer` is the command buffer that the pipeline will be bound to.
- `pipelineBindPoint` is a `VkPipelineBindPoint` value specifying whether to bind to the compute or graphics bind point. Binding one does not disturb the other.
- `pipeline` is the pipeline to be bound.

Once bound, a pipeline binding affects subsequent graphics or compute commands in the command buffer until a different pipeline is bound to the bind point. The pipeline bound to `VK_PIPELINE_BIND_POINT_COMPUTE` controls the behavior of `vkCmdDispatch` and `vkCmdDispatchIndirect`. The pipeline bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` controls the behavior of all drawing commands. The pipeline bound to `VK_PIPELINE_BIND_POINT_RAY_TRACING_NV` controls the behavior of `vkCmdTraceRaysNV`. No other commands are affected by the pipeline state.

## Valid Usage

- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, the `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, the `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, `pipeline` **must** be a compute pipeline
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, `pipeline` **must** be a graphics pipeline
- If the `variable multisample rate` feature is not supported, `pipeline` is a graphics pipeline, the current subpass has no attachments, and this is not the first call to this function with a graphics pipeline after transitioning to the current subpass, then the sample count specified by this pipeline **must** match that set in the previous pipeline
- If `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE`, and `pipeline` is a graphics pipeline created with a `VkPipelineSampleLocationsStateCreateInfoEXT` structure having its `sampleLocationsEnable` member set to `VK_TRUE` but without `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT` enabled then the current render pass instance **must** have been begun by specifying a `VkRenderPassSampleLocationsBeginInfoEXT` structure whose `pPostSubpassSampleLocations` member contains an element with a `subpassIndex` matching the current subpass index and the `sampleLocationsInfo` member of that element **must** match the `sampleLocationsInfo` specified in `VkPipelineSampleLocationsStateCreateInfoEXT` when the pipeline was created
- This command **must** not be recorded when transform feedback is active
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_RAY_TRACING_NV`, the `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_RAY_TRACING_NV`, the `pipeline` **must** be a ray tracing pipeline

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pipelineBindPoint` **must** be a valid `VkPipelineBindPoint` value
- `pipeline` **must** be a valid `VkPipeline` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- Both of `commandBuffer`, and `pipeline` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

Possible values of `vkCmdBindPipeline::pipelineBindPoint`, specifying the bind point of a pipeline object, are:

```
typedef enum VkPipelineBindPoint {
    VK_PIPELINE_BIND_POINT_GRAPHICS = 0,
    VK_PIPELINE_BIND_POINT_COMPUTE = 1,
    VK_PIPELINE_BIND_POINT_RAY_TRACING_NV = 1000165000,
    VK_PIPELINE_BIND_POINT_MAX_ENUM = 0x7FFFFFFF
} VkPipelineBindPoint;
```

- `VK_PIPELINE_BIND_POINT_COMPUTE` specifies binding as a compute pipeline.
- `VK_PIPELINE_BIND_POINT_GRAPHICS` specifies binding as a graphics pipeline.
- `VK_PIPELINE_BIND_POINT_RAY_TRACING_NV` specifies binding as a ray tracing pipeline.

## 9.9. Dynamic State

When a pipeline object is bound, any pipeline object state that is not specified as dynamic is applied to the command buffer state. Pipeline object state that is specified as dynamic is not applied to the command buffer state at this time. Instead, dynamic state **can** be modified at any time and persists for the lifetime of the command buffer, or until modified by another dynamic state setting command or another pipeline bind.

When a pipeline object is bound, the following applies to each state parameter:

- If the state is not specified as dynamic in the new pipeline object, then that command buffer state is overwritten by the state in the new pipeline object.
- If the state is specified as dynamic in both the new and the previous pipeline object, then that command buffer state is not disturbed.
- If the state is specified as dynamic in the new pipeline object but is not specified as dynamic in the previous pipeline object, then that command buffer state becomes undefined. If the state is

an array, then the entire array becomes undefined.

- If the state is an array specified as dynamic in both the new and the previous pipeline object, and the array size is not the same in both pipeline objects, then that command buffer state becomes undefined.

Dynamic state setting commands **must** not be issued for state that is not specified as dynamic in the bound pipeline object.

Dynamic state that does not affect the result of operations **can** be left undefined.

*Note*



For example, if blending is disabled by the pipeline object state then the dynamic color blend constants do not need to be specified in the command buffer, even if this state is specified as dynamic in the pipeline object.

## 9.10. Pipeline Shader Information

When a pipeline is created, its state and shaders are compiled into zero or more device-specific executables, which are used when executing commands against that pipeline. To query the properties of these executables, call:

```
VkResult vkGetPipelineExecutablePropertiesKHR(  
    VkDevice                                     device,  
    const VkPipelineInfoKHR*                      pPipelineInfo,  
    uint32_t*                                     pExecutableCount,  
    VkPipelineExecutablePropertiesKHR*            pProperties);
```

- `device` is the device that created the pipeline.
- `pPipelineInfo` describes the pipeline being queried.
- `pExecutableCount` is a pointer to an integer related to the number of pipeline executables available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkPipelineExecutablePropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of executables associated with the pipeline is returned in `pExecutableCount`. Otherwise, `pExecutableCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If `pExecutableCount` is less than the number of executables associated with the pipeline, at most `pExecutableCount` structures will be written and `vkGetPipelineExecutablePropertiesKHR` will return `VK_INCOMPLETE`.

## Valid Usage

- `pipelineExecutableInfo` **must** be enabled.
- `pipeline` member of `pPipelineInfo` **must** have been created with `device`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pPipelineInfo` **must** be a valid pointer to a valid `VkPipelineInfoKHR` structure
- `pExecutableCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pExecutableCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pExecutableCount` `VkPipelineExecutablePropertiesKHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkPipelineInfoKHR` structure is defined as:

```
typedef struct VkPipelineInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkPipeline         pipeline;
} VkPipelineInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pipeline` is a `VkPipeline` handle.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_INFO_KHR`
- `pNext` must be `NULL`
- `pipeline` must be a valid `VkPipeline` handle

The `VkPipelineExecutablePropertiesKHR` structure is defined as:

```
typedef struct VkPipelineExecutablePropertiesKHR {
    VkStructureType      sType;
    void*                pNext;
    VkShaderStageFlags   stages;
    char                 name[VK_MAX_DESCRIPTION_SIZE];
    char                 description[VK_MAX_DESCRIPTION_SIZE];
    uint32_t              subgroupSize;
} VkPipelineExecutablePropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `stages` is a bitmask of `VkShaderStageFlagBits` indicating which shader stages (if any) were principally used as inputs to compile this pipeline executable.
- `name` is an array of `VK_MAX_DESCRIPTION_SIZE` `char` containing a null-terminated UTF-8 string which is a short human readable name for this executable.
- `description` is an array of `VK_MAX_DESCRIPTION_SIZE` `char` containing a null-terminated UTF-8 string which is a human readable description for this executable.
- `subgroupSize` is the subgroup size with which this executable is dispatched.

The `stages` field **may** be zero or it **may** contain one or more bits describing the stages principally used to compile this pipeline. Not all implementations have a 1:1 mapping between shader stages and pipeline executables and some implementations **may** reduce a given shader stage to fixed function hardware programming such that no executable is available. No guarantees are provided about the mapping between shader stages and pipeline executables and `stages` **should** be considered a best effort hint. Because the application **cannot** rely on the `stages` field to provide an exact description, `name` and `description` provide a human readable name and description which more accurately describes the given pipeline executable.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_PROPERTIES_KHR`
- `pNext` must be `NULL`

Each pipeline executable **may** have a set of statistics associated with it that are generated by the

pipeline compilation process. These statistics **may** include things such as instruction counts, amount of spilling (if any), maximum number of simultaneous threads, or anything else which **may** aid developers in evaluating the expected performance of a shader. To query the compile-time statistics associated with a pipeline executable, call:

```
VkResult vkGetPipelineExecutableStatisticsKHR(  
    VkDevice device,  
    const VkPipelineExecutableInfoKHR* pExecutableInfo,  
    uint32_t* pStatisticCount,  
    VkPipelineExecutableStatisticKHR* pStatistics);
```

- `device` is the device that created the pipeline.
- `pExecutableInfo` describes the pipeline executable being queried.
- `pStatisticCount` is a pointer to an integer related to the number of statistics available or queried, as described below.
- `pStatistics` is either `NULL` or a pointer to an array of `VkPipelineExecutableStatisticKHR` structures.

If `pStatistics` is `NULL`, then the number of statistics associated with the pipeline executable is returned in `pStatisticCount`. Otherwise, `pStatisticCount` **must** point to a variable set by the user to the number of elements in the `pStatistics` array, and on return the variable is overwritten with the number of structures actually written to `pStatistics`. If `pStatisticCount` is less than the number of statistics associated with the pipeline executable, at most `pStatisticCount` structures will be written and `vkGetPipelineExecutableStatisticsKHR` will return `VK_INCOMPLETE`.

## Valid Usage

- `pipelineExecutableInfo` **must** be enabled.
- `pipeline` member of `pExecutableInfo` **must** have been created with `device`.
- `pipeline` member of `pExecutableInfo` **must** have been created with `VK_PIPELINE_CREATE_CAPTURE_STATISTICS_BIT_KHR` set in the `flags` field of `VkGraphicsPipelineCreateInfo` or `VkComputePipelineCreateInfo`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pExecutableInfo` **must** be a valid pointer to a valid `VkPipelineExecutableInfoKHR` structure
- `pStatisticCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pStatisticCount` is not `0`, and `pStatistics` is not `NULL`, `pStatistics` **must** be a valid pointer to an array of `pStatisticCount` `VkPipelineExecutableStatisticKHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkPipelineExecutableInfoKHR` structure is defined as:

```
typedef struct VkPipelineExecutableInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkPipeline         pipeline;
    uint32_t           executableIndex;
} VkPipelineExecutableInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pipeline` is the pipeline to query.
- `executableIndex` is the index of the executable to query in the array of executable properties returned by `vkGetPipelineExecutablePropertiesKHR`.

### Valid Usage

- `executableIndex` **must** be less than the number of executables associated with `pipeline` as returned in the `pExecutableCount` parameter of `vkGetPipelineExecutablePropertiesKHR`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INFO_KHR`
- `pNext` **must** be `NULL`
- `pipeline` **must** be a valid `VkPipeline` handle

The `VkPipelineExecutableStatisticKHR` structure is defined as:

```

typedef struct VkPipelineExecutableStatisticKHR {
    VkStructureType sType;
    void* pNext;
    char name[VK_MAX_DESCRIPTION_SIZE];
    char description[VK_MAX_DESCRIPTION_SIZE];
    VkPipelineExecutableStatisticFormatKHR format;
    VkPipelineExecutableStatisticValueKHR value;
} VkPipelineExecutableStatisticKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **name** is an array of **VK\_MAX\_DESCRIPTION\_SIZE** **char** containing a null-terminated UTF-8 string which is a short human readable name for this statistic.
- **description** is an array of **VK\_MAX\_DESCRIPTION\_SIZE** **char** containing a null-terminated UTF-8 string which is a human readable description for this statistic.
- **format** is a **VkPipelineExecutableStatisticFormatKHR** value specifying the format of the data found in **value**.
- **value** is the value of this statistic.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_PIPELINE\_EXECUTABLE\_STATISTIC\_KHR**
- **pNext** **must** be **NULL**

The **VkPipelineExecutableStatisticFormatKHR** enum is defined as:

```

typedef enum VkPipelineExecutableStatisticFormatKHR {
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_BOOL32_KHR = 0,
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_INT64_KHR = 1,
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_UINT64_KHR = 2,
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_FLOAT64_KHR = 3,
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_MAX_ENUM_KHR = 0x7FFFFFFF
} VkPipelineExecutableStatisticFormatKHR;

```

- **VK\_PIPELINE\_EXECUTABLE\_STATISTIC\_FORMAT\_BOOL32\_KHR** specifies that the statistic is returned as a 32-bit boolean value which **must** be either **VK\_TRUE** or **VK\_FALSE** and **should** be read from the **b32** field of **VkPipelineExecutableStatisticValueKHR**.
- **VK\_PIPELINE\_EXECUTABLE\_STATISTIC\_FORMAT\_INT64\_KHR** specifies that the statistic is returned as a signed 64-bit integer and **should** be read from the **i64** field of **VkPipelineExecutableStatisticValueKHR**.
- **VK\_PIPELINE\_EXECUTABLE\_STATISTIC\_FORMAT\_UINT64\_KHR** specifies that the statistic is returned as an unsigned 64-bit integer and **should** be read from the **u64** field of **VkPipelineExecutableStatisticValueKHR**.

- `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_FLOAT64_KHR` specifies that the statistic is returned as a 64-bit floating-point value and **should** be read from the `f64` field of `VkPipelineExecutableStatisticValueKHR`.

The `VkPipelineExecutableStatisticValueKHR` union is defined as:

```
typedef union VkPipelineExecutableStatisticValueKHR {
    VkBool32    b32;
    int64_t     i64;
    uint64_t    u64;
    double      f64;
} VkPipelineExecutableStatisticValueKHR;
```

- `b32` is the 32-bit boolean value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_BOOL32_KHR`.
- `i64` is the signed 64-bit integer value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_INT64_KHR`.
- `u64` is the unsigned 64-bit integer value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_UINT64_KHR`.
- `f64` is the 64-bit floating-point value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_FLOAT64_KHR`.

Each pipeline executable **may** have one or more text or binary internal representations associated with it which are generated as part of the compile process. These **may** include the final shader assembly, a binary form of the compiled shader, or the shader compiler's internal representation at any number of intermediate compile steps. To query the internal representations associated with a pipeline executable, call:

```
VkResult vkGetPipelineExecutableInternalRepresentationsKHR(
    VkDevice                                     device,
    const VkPipelineExecutableInfoKHR*          pExecutableInfo,
    uint32_t*                                    pInternalRepresentationCount,
    VkPipelineExecutableInternalRepresentationKHR* pInternalRepresentations);
```

- `device` is the device that created the pipeline.
- `pExecutableInfo` describes the pipeline executable being queried.
- `pInternalRepresentationCount` is a pointer to an integer related to the number of internal representations available or queried, as described below.
- `pInternalRepresentations` is either `NULL` or a pointer to an array of `VkPipelineExecutableInternalRepresentationKHR` structures.

If `pInternalRepresentations` is `NULL`, then the number of internal representations associated with the pipeline executable is returned in `pInternalRepresentationCount`. Otherwise, `pInternalRepresentationCount` **must** point to a variable set by the user to the number of elements in the `pInternalRepresentations` array, and on return the variable is overwritten with the number of

structures actually written to `pInternalRepresentations`. If `pInternalRepresentationCount` is less than the number of internal representations associated with the pipeline executable, at most `pInternalRepresentationCount` structures will be written and `vkGetPipelineExecutableInternalRepresentationsKHR` will return `VK_INCOMPLETE`.

While the details of the internal representations remain implementation dependent, the implementation **should** order the internal representations in the order in which they occur in the compile pipeline with the final shader assembly (if any) last.

## Valid Usage

- `pipelineExecutableInfo` **must** be enabled.
- `pipeline` member of `pExecutableInfo` **must** have been created with `device`.
- `pipeline` member of `pExecutableInfo` **must** have been created with `VK_PIPELINE_CREATE_CAPTURE_INTERNAL REPRESENTATIONS_BIT_KHR` set in the `flags` field of `VkGraphicsPipelineCreateInfo` or `VkComputePipelineCreateInfo`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pExecutableInfo` **must** be a valid pointer to a valid `VkPipelineExecutableCreateInfoKHR` structure
- `pInternalRepresentationCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pInternalRepresentationCount` is not `0`, and `pInternalRepresentations` is not `NULL`, `pInternalRepresentations` **must** be a valid pointer to an array of `pInternalRepresentationCount` `VkPipelineExecutableInternalRepresentationKHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkPipelineExecutableInternalRepresentationKHR` structure is defined as:

```

typedef struct VkPipelineExecutableInternalRepresentationKHR {
    VkStructureType    sType;
    void*           pNext;
    char             name[VK_MAX_DESCRIPTION_SIZE];
    char             description[VK_MAX_DESCRIPTION_SIZE];
    VkBool32        isText;
    size_t          dataSize;
    void*           pData;
} VkPipelineExecutableInternalRepresentationKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **name** is an array of **VK\_MAX\_DESCRIPTION\_SIZE** **char** containing a null-terminated UTF-8 string which is a short human readable name for this internal representation.
- **description** is an array of **VK\_MAX\_DESCRIPTION\_SIZE** **char** containing a null-terminated UTF-8 string which is a human readable description for this internal representation.
- **isText** specifies whether the returned data is text or opaque data. If **isText** is **VK\_TRUE** then the data returned in **pData** is text and is guaranteed to be a null-terminated UTF-8 string.
- **dataSize** is an integer related to the size, in bytes, of the internal representation data, as described below.
- **pData** is either **NULL** or a pointer to an block of data into which the implementation will write the textual form of the internal representation.

If **pData** is **NULL**, then the size, in bytes, of the internal representation data is returned in **dataSize**. Otherwise, **dataSize** must be the size of the buffer, in bytes, pointed to by **pData** and on return **dataSize** is overwritten with the number of bytes of data actually written to **pData** including any trailing null character. If **dataSize** is less than the size, in bytes, of the internal representation data, at most **dataSize** bytes of data will be written to **pData** and **vkGetPipelineExecutableInternalRepresentationsKHR** will return **VK\_INCOMPLETE**. If **isText** is **VK\_TRUE** and **pData** is not **NULL** and **dataSize** is not zero, the last byte written to **pData** will be a null character.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_PIPELINE\_EXECUTABLE\_INTERNAL\_REPRESENTATION\_KHR**
- **pNext** **must** be **NULL**
- **name** **must** be a null-terminated UTF-8 string whose length is less than or equal to **VK\_MAX\_DESCRIPTION\_SIZE**
- **description** **must** be a null-terminated UTF-8 string whose length is less than or equal to **VK\_MAX\_DESCRIPTION\_SIZE**
- If **dataSize** is not **0**, and **pData** is not **NULL**, **pData** **must** be a valid pointer to an array of **dataSize** bytes

Information about a particular shader that has been compiled as part of a pipeline object can be

extracted by calling:

```
VkResult vkGetShaderInfoAMD(  
    VkDevice device,  
    VkPipeline pipeline,  
    VkShaderStageFlagBits shaderStage,  
    VkShaderInfoTypeAMD infoType,  
    size_t* pInfoSize,  
    void* pInfo);
```

- `device` is the device that created `pipeline`.
- `pipeline` is the target of the query.
- `shaderStage` identifies the particular shader within the pipeline about which information is being queried.
- `infoType` describes what kind of information is being queried.
- `pInfoSize` is a pointer to a value related to the amount of data the query returns, as described below.
- `pInfo` is either `NULL` or a pointer to a buffer.

If `pInfo` is `NULL`, then the maximum size of the information that **can** be retrieved about the shader, in bytes, is returned in `pInfoSize`. Otherwise, `pInfoSize` **must** point to a variable set by the user to the size of the buffer, in bytes, pointed to by `pInfo`, and on return the variable is overwritten with the amount of data actually written to `pInfo`.

If `pInfoSize` is less than the maximum size that **can** be retrieved by the pipeline cache, then at most `pInfoSize` bytes will be written to `pInfo`, and `vkGetShaderInfoAMD` will return `VK_INCOMPLETE`.

Not all information is available for every shader and implementations may not support all kinds of information for any shader. When a certain type of information is unavailable, the function returns `VK_ERROR_FEATURE_NOT_PRESENT`.

If information is successfully and fully queried, the function will return `VK_SUCCESS`.

For `infoType VK_SHADER_INFO_TYPE_STATISTICS_AMD`, an instance of `VkShaderStatisticsInfoAMD` will be written to the buffer pointed to by `pInfo`. This structure will be populated with statistics regarding the physical device resources used by that shader along with other miscellaneous information and is described in further detail below.

For `infoType VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD`, `pInfo` is a pointer to a UTF-8 null-terminated string containing human-readable disassembly. The exact formatting and contents of the disassembly string are vendor-specific.

The formatting and contents of all other types of information, including `infoType VK_SHADER_INFO_TYPE_BINARY_AMD`, are left to the vendor and are not further specified by this extension.

## Valid Usage (Implicit)

- **device** **must** be a valid `VkDevice` handle
- **pipeline** **must** be a valid `VkPipeline` handle
- **shaderStage** **must** be a valid `VkShaderStageFlagBits` value
- **infoType** **must** be a valid `VkShaderInfoTypeAMD` value
- **pInfoSize** **must** be a valid pointer to a `size_t` value
- If the value referenced by **pInfoSize** is not `0`, and **pInfo** is not `NULL`, **pInfo** **must** be a valid pointer to an array of **pInfoSize** bytes
- **pipeline** **must** have been created, allocated, or retrieved from **device**

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_FEATURE_NOT_PRESENT`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

Possible values of `vkGetShaderInfoAMD::infoType`, specifying the information being queried from a shader, are:

```
typedef enum VkShaderInfoTypeAMD {
    VK_SHADER_INFO_TYPE_STATISTICS_AMD = 0,
    VK_SHADER_INFO_TYPE_BINARY_AMD = 1,
    VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD = 2,
    VK_SHADER_INFO_TYPE_MAX_ENUM_AMD = 0x7FFFFFFF
} VkShaderInfoTypeAMD;
```

- `VK_SHADER_INFO_TYPE_STATISTICS_AMD` specifies that device resources used by a shader will be queried.
- `VK_SHADER_INFO_TYPE_BINARY_AMD` specifies that implementation-specific information will be queried.
- `VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD` specifies that human-readable disassembly of a shader.

The `VkShaderStatisticsInfoAMD` structure is defined as:

```

typedef struct VkShaderStatisticsInfoAMD {
    VkShaderStageFlags          shaderStageMask;
    VkShaderResourceUsageAMD   resourceUsage;
    uint32_t                   numPhysicalVgprs;
    uint32_t                   numPhysicalSgprs;
    uint32_t                   numAvailableVgprs;
    uint32_t                   numAvailableSgprs;
    uint32_t                   computeWorkGroupSize[3];
} VkShaderStatisticsInfoAMD;

```

- **shaderStageMask** are the combination of logical shader stages contained within this shader.
- **resourceUsage** is an instance of [VkShaderResourceUsageAMD](#) describing internal physical device resources used by this shader.
- **numPhysicalVgprs** is the maximum number of vector instruction general-purpose registers (VGPRs) available to the physical device.
- **numPhysicalSgprs** is the maximum number of scalar instruction general-purpose registers (SGPRs) available to the physical device.
- **numAvailableVgprs** is the maximum limit of VGPRs made available to the shader compiler.
- **numAvailableSgprs** is the maximum limit of SGPRs made available to the shader compiler.
- **computeWorkGroupSize** is the local workgroup size of this shader in { X, Y, Z } dimensions.

Some implementations may merge multiple logical shader stages together in a single shader. In such cases, **shaderStageMask** will contain a bitmask of all of the stages that are active within that shader. Consequently, if specifying those stages as input to [vkGetShaderInfoAMD](#), the same output information **may** be returned for all such shader stage queries.

The number of available VGPRs and SGPRs (**numAvailableVgprs** and **numAvailableSgprs** respectively) are the shader-addressable subset of physical registers that is given as a limit to the compiler for register assignment. These values **may** further be limited by implementations due to performance optimizations where register pressure is a bottleneck.

The [VkShaderResourceUsageAMD](#) structure is defined as:

```

typedef struct VkShaderResourceUsageAMD {
    uint32_t      numUsedVgprs;
    uint32_t      numUsedSgprs;
    uint32_t      ldsSizePerLocalWorkGroup;
    size_t        ldsUsageSizeInBytes;
    size_t        scratchMemUsageInBytes;
} VkShaderResourceUsageAMD;

```

- **numUsedVgprs** is the number of vector instruction general-purpose registers used by this shader.
- **numUsedSgprs** is the number of scalar instruction general-purpose registers used by this shader.
- **ldsSizePerLocalWorkGroup** is the maximum local data store size per work group in bytes.

- `ldsUsageSizeInBytes` is the LDS usage size in bytes per work group by this shader.
- `scratchMemUsageInBytes` is the scratch memory usage in bytes by this shader.

## 9.11. Pipeline Compiler Control

The compilation of a pipeline **can** be tuned by including a `VkPipelineCompilerControlCreateInfoAMD` structure in the `pNext` chain of `VkGraphicsPipelineCreateInfo` or `VkComputePipelineCreateInfo`.

```
typedef struct VkPipelineCompilerControlCreateInfoAMD {
    VkStructureType           sType;
    const void*                pNext;
    VkPipelineCompilerControlFlagsAMD compilerControlFlags;
} VkPipelineCompilerControlCreateInfoAMD;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `compilerControlFlags` is a bitmask of `VkPipelineCompilerControlFlagBitsAMD` affecting how the pipeline will be compiled.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COMPILER_CONTROL_CREATE_INFO_AMD`
- `compilerControlFlags` **must** be `0`

There are currently no available flags for this extension; flags will be added by future versions of this extension.

```
typedef enum VkPipelineCompilerControlFlagBitsAMD {
    VK_PIPELINE_COMPILER_CONTROL_FLAG_BITS_MAX_ENUM_AMD = 0x7FFFFFFF
} VkPipelineCompilerControlFlagBitsAMD;
```

## 9.12. Ray Tracing Pipeline

Ray tracing pipelines consist of multiple shader stages, fixed-function traversal stages, and a pipeline layout.

To create ray tracing pipelines, call:

```
VkResult vkCreateRayTracingPipelinesNV(  
    VkDevice device,  
    VkPipelineCache pipelineCache,  
    uint32_t createInfoCount,  
    const VkRayTracingPipelineCreateInfoNV* pCreateInfos,  
    const VkAllocationCallbacks* pAllocator,  
    VkPipeline* pPipelines);
```

- `device` is the logical device that creates the ray tracing pipelines.
- `pipelineCache` is either `VK_NULL_HANDLE`, indicating that pipeline caching is disabled, or the handle of a valid `pipeline cache` object, in which case use of that cache is enabled for the duration of the command.
- `createInfoCount` is the length of the `pCreateInfos` and `pPipelines` arrays.
- `pCreateInfos` is a pointer to an array of `VkRayTracingPipelineCreateInfoNV` structures.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pPipelines` is a pointer to an array in which the resulting ray tracing pipeline objects are returned.

## Valid Usage

- If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and the `basePipelineIndex` member of that same element is not `-1`, `basePipelineIndex` **must** be less than the index into `pCreateInfos` that corresponds to that element
- If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, the base pipeline **must** have been created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` flag set

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `pipelineCache` is not `VK_NULL_HANDLE`, `pipelineCache` **must** be a valid `VkPipelineCache` handle
- `pCreateInfos` **must** be a valid pointer to an array of `createInfoCount` valid `VkRayTracingPipelineCreateInfoNV` structures
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pPipelines` **must** be a valid pointer to an array of `createInfoCount` `VkPipeline` handles
- `createInfoCount` **must** be greater than `0`
- If `pipelineCache` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_SHADER_NV`

The `VkRayTracingPipelineCreateInfoNV` structure is defined as:

```
typedef struct VkRayTracingPipelineCreateInfoNV {
    VkStructureType                         sType;
    const void*                             pNext;
    VkPipelineCreateFlags                   flags;
    uint32_t                               stageCount;
    const VkPipelineShaderStageCreateInfo* pStages;
    uint32_t                               groupCount;
    const VkRayTracingShaderGroupCreateInfoNV* pGroups;
    uint32_t                               maxRecursionDepth;
    VkPipelineLayout                        layout;
    VkPipeline                            basePipelineHandle;
    int32_t                                basePipelineIndex;
} VkRayTracingPipelineCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkPipelineCreateFlagBits` specifying how the pipeline will be generated.

- `stageCount` is the number of entries in the `pStages` array.
- `pStages` is a pointer to an array of `stageCount` `VkPipelineShaderStageCreateInfo` structures describing the set of the shader stages to be included in the ray tracing pipeline.
- `groupCount` is the number of entries in the `pGroups` array.
- `pGroups` is a pointer to an array of `groupCount` `VkRayTracingShaderGroupCreateInfoNV` structures describing the set of the shader stages to be included in each shader group in the ray tracing pipeline.
- `maxRecursionDepth` is the maximum recursion that will be called from this pipeline.
- `layout` is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
- `basePipelineHandle` is a pipeline to derive from.
- `basePipelineIndex` is an index into the `pCreateInfos` parameter to use as a pipeline to derive from.

The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in [Pipeline Derivatives](#).

## Valid Usage

- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is `-1`, `basePipelineHandle` **must** be a valid handle to a ray tracing `VkPipeline`
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` **must** be a valid index into the calling command's `pCreateInfos` parameter
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not `-1`, `basePipelineHandle` **must** be `VK_NULL_HANDLE`
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` **must** be `-1`
- The `stage` member of one element of `pStages` **must** be `VK_SHADER_STAGE_RAYGEN_BIT_NV`
- The shader code for the entry points identified by `pStages`, and the rest of the state identified by this structure **must** adhere to the pipeline linking rules described in the [Shader Interfaces](#) chapter
- `layout` **must** be consistent with all shaders specified in `pStages`
- The number of resources in `layout` accessible to each shader stage that is used by the pipeline **must** be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`
- `maxRecursionDepth` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxRecursionDepth`

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_NV`
- **pNext** must be `NULL` or a pointer to a valid instance of `VkPipelineCreationFeedbackCreateInfoEXT`
- **flags** must be a valid combination of `VkPipelineCreateFlagBits` values
- **pStages** must be a valid pointer to an array of **stageCount** valid `VkPipelineShaderStageCreateInfo` structures
- **pGroups** must be a valid pointer to an array of **groupCount** valid `VkRayTracingShaderGroupCreateInfoNV` structures
- **layout** must be a valid `VkPipelineLayout` handle
- **stageCount** must be greater than `0`
- **groupCount** must be greater than `0`
- Both of `basePipelineHandle`, and `layout` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

The `VkRayTracingShaderGroupCreateInfoNV` structure is defined as:

```
typedef struct VkRayTracingShaderGroupCreateInfoNV {
    VkStructureType           sType;
    const void*               pNext;
    VkRayTracingShaderGroupTypeNV type;
    uint32_t                  generalShader;
    uint32_t                  closestHitShader;
    uint32_t                  anyHitShader;
    uint32_t                  intersectionShader;
} VkRayTracingShaderGroupCreateInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **type** is the type of hit group specified in this structure.
- **generalShader** is the index of the ray generation, miss, or callable shader from `VkRayTracingPipelineCreateInfoNV::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV` and `VK_SHADER_UNUSED_NV` otherwise.
- **closestHitShader** is the optional index of the closest hit shader from `VkRayTracingPipelineCreateInfoNV::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV` or `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_NV` and `VK_SHADER_UNUSED_NV` otherwise.
- **anyHitShader** is the optional index of the any-hit shader from `VkRayTracingPipelineCreateInfoNV ::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV` or

`VK_RAY_TRACING_SHADER_GROUP_TYPE PROCEDURAL_HIT_GROUP_NV` and `VK_SHADER_UNUSED_NV` otherwise.

- `intersectionShader` is the index of the intersection shader from `VkRayTracingPipelineCreateInfoNV::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE PROCEDURAL_HIT_GROUP_NV` and `VK_SHADER_UNUSED_NV` otherwise.

## Valid Usage

- If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV` then `generalShader` **must** be a valid index into `pStages` referring to a shader of `VK_SHADER_STAGE_RAYGEN_BIT_NV`, `VK_SHADER_STAGE_MISS_BIT_NV`, or `VK_SHADER_STAGE_CALLABLE_BIT_NV`
- If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV` then `closestHitShader`, `anyHitShader`, and `intersectionShader` **must** be `VK_SHADER_UNUSED_NV`
- If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE PROCEDURAL_HIT_GROUP_NV` then `intersectionShader` **must** be a valid index into `pStages` referring to a shader of `VK_SHADER_STAGE_INTERSECTION_BIT_NV`
- If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV` then `intersectionShader` **must** be `VK_SHADER_UNUSED_NV`
- `closestHitShader` **must** be either `VK_SHADER_UNUSED_NV` or a valid index into `pStages` referring to a shader of `VK_SHADER_STAGE_CLOSEST_HIT_BIT_NV`
- `anyHitShader` **must** be either `VK_SHADER_UNUSED_NV` or a valid index into `pStages` referring to a shader of `VK_SHADER_STAGE_ANY_HIT_BIT_NV`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_NV`
- `pNext` **must** be `NULL`
- `type` **must** be a valid `VkRayTracingShaderGroupTypeNV` value

Possible values of `type` in `VkRayTracingShaderGroupCreateInfoNV` are:

```
typedef enum VkRayTracingShaderGroupTypeNV {
    VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV = 0,
    VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV = 1,
    VK_RAY_TRACING_SHADER_GROUP_TYPE PROCEDURAL_HIT_GROUP_NV = 2,
    VK_RAY_TRACING_SHADER_GROUP_TYPE_MAX_ENUM_NV = 0x7FFFFFFF
} VkRayTracingShaderGroupTypeNV;
```

- `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV` indicates a shader group with a single `VK_SHADER_STAGE_RAYGEN_BIT_NV`, `VK_SHADER_STAGE_MISS_BIT_NV`, or `VK_SHADER_STAGE_CALLABLE_BIT_NV` shader in it.
- `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV` specifies a shader group that only hits triangles and **must** not contain an intersection shader, only closest hit and any-hit.

- `VK_RAY_TRACING_SHADER_GROUP_TYPE PROCEDURAL_HIT_GROUP_NV` specifies a shader group that only intersects with custom geometry and **must** contain an intersection shader and **may** contain closest hit and any-hit shaders.

*Note*



For current group types, the hit group type could be inferred from the presence or absence of the intersection shader, but we provide the type explicitly for future hit groups that do not have that property.

To query the opaque handles of shaders in the ray tracing pipeline, call:

```
VkResult vkGetRayTracingShaderGroupHandlesNV(
    VkDevice                                     device,
    VkPipeline                                    pipeline,
    uint32_t                                     firstGroup,
    uint32_t                                     groupCount,
    size_t                                         dataSize,
    void*                                         pData);
```

- `device` is the logical device containing the ray tracing pipeline.
- `pipeline` is the ray tracing pipeline object containing the shaders.
- `firstGroup` is the index of the first group to retrieve a handle for from the `VkRayTracingShaderGroupCreateInfoNV::pGroups` array.
- `groupCount` is the number of shader handles to retrieve.
- `dataSize` is the size in bytes of the buffer pointed to by `pData`.
- `pData` is a pointer to a user-allocated buffer where the results will be written.

## Valid Usage

- The sum of `firstGroup` and `groupCount` **must** be less than the number of shader groups in `pipeline`.
- `dataSize` **must** be at least `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupHandleSize`  $\times$  `groupCount`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pipeline` **must** be a valid `VkPipeline` handle
- `pData` **must** be a valid pointer to an array of `dataSize` bytes
- `dataSize` **must** be greater than 0
- `pipeline` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Ray tracing pipelines **can** contain more shaders than a graphics or compute pipeline, so to allow parallel compilation of shaders within a pipeline, an application **can** choose to defer compilation until a later point in time.

To compile a deferred shader in a pipeline call:

```
VkResult vkCompileDeferredNV(  
    VkDevice device,  
    VkPipeline pipeline,  
    uint32_t shader);
```

- `device` is the logical device containing the ray tracing pipeline.
- `pipeline` is the ray tracing pipeline object containing the shaders.
- `shader` is the index of the shader to compile.

### Valid Usage

- `pipeline` **must** have been created with `VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV`
- `shader` **must** not have been called as a deferred compile before

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pipeline` **must** be a valid `VkPipeline` handle
- `pipeline` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

## 9.13. Pipeline Creation Feedback

Feedback about the creation of a particular pipeline object **can** be obtained by including a `VkPipelineCreationFeedbackCreateInfoEXT` structure in the `pNext` chain of `VkGraphicsPipelineCreateInfo`, `VkRayTracingPipelineCreateInfoNV`, or `VkComputePipelineCreateInfo`. The `VkPipelineCreationFeedbackCreateInfoEXT` structure is defined as:

```
typedef struct VkPipelineCreationFeedbackCreateInfoEXT {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineCreationFeedbackEXT* pPipelineCreationFeedback;
    uint32_t                  pipelineStageCreationFeedbackCount;
    VkPipelineCreationFeedbackEXT* pPipelineStageCreationFeedbacks;
} VkPipelineCreationFeedbackCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pPipelineCreationFeedback` is a pointer to a `VkPipelineCreationFeedbackEXT` structure.
- `pipelineStageCreationFeedbackCount` is the number of elements in `pPipelineStageCreationFeedbacks`.
- `pPipelineStageCreationFeedbacks` is a pointer to an array of `pipelineStageCreationFeedbackCount` `VkPipelineCreationFeedbackEXT` structures.

An implementation **should** write pipeline creation feedback to `pPipelineCreationFeedback` and **may** write pipeline stage creation feedback to `pPipelineStageCreationFeedbacks`. An implementation **must** set or clear the `VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT` in `VkPipelineCreationFeedbackEXT::flags` for `pPipelineCreationFeedback` and every element of `pPipelineStageCreationFeedbacks`.

### Note



One common scenario for an implementation to skip per-stage feedback is when `VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT_EXT` is set in `pPipelineCreationFeedback`.

When chained to `VkRayTracingPipelineCreateInfoNV` or `VkGraphicsPipelineCreateInfo`, the `i` element of `pPipelineStageCreationFeedbacks` corresponds to the `i` element of `VkRayTracingPipelineCreateInfoNV::pStages` or `VkGraphicsPipelineCreateInfo::pStages`. When chained to `VkComputePipelineCreateInfo`, the first element of `pPipelineStageCreationFeedbacks` corresponds to `VkComputePipelineCreateInfo::stage`.

## Valid Usage

- When chained to `VkGraphicsPipelineCreateInfo`, `VkPipelineCreationFeedbackEXT::pipelineStageCreationFeedbackCount` **must** equal `VkGraphicsPipelineCreateInfo::stageCount`
- When chained to `VkComputePipelineCreateInfo`, `VkPipelineCreationFeedbackEXT::pipelineStageCreationFeedbackCount` **must** equal 1
- When chained to `VkRayTracingPipelineCreateInfoNV`, `VkPipelineCreationFeedbackEXT::pipelineStageCreationFeedbackCount` **must** equal `VkRayTracingPipelineCreateInfoNV::stageCount`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO_EXT`
- `pPipelineCreationFeedback` **must** be a valid pointer to a `VkPipelineCreationFeedbackEXT` structure
- `pPipelineStageCreationFeedbacks` **must** be a valid pointer to an array of `pipelineStageCreationFeedbackCount` `VkPipelineCreationFeedbackEXT` structures
- `pipelineStageCreationFeedbackCount` **must** be greater than 0

The `VkPipelineCreationFeedbackEXT` structure is defined as:

```
typedef struct VkPipelineCreationFeedbackEXT {
    VkPipelineCreationFeedbackFlagsEXT    flags;
    uint64_t                            duration;
} VkPipelineCreationFeedbackEXT;
```

- `flags` is a bitmask of `VkPipelineCreationFeedbackFlagBitsEXT` providing feedback about the creation of a pipeline or of a pipeline stage.
- `duration` is the duration spent creating a pipeline or pipeline stage in nanoseconds.

If the `VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT` is not set in `flags`, an implementation **must** not set any other bits in `flags`, and all other `VkPipelineCreationFeedbackEXT` data members are undefined.

Possible values of the `flags` member of `VkPipelineCreationFeedbackEXT` are:

```

typedef enum VkPipelineCreationFeedbackFlagBitsEXT {
    VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT = 0x00000001,
    VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT_EXT = 0x00000002,
    VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT_EXT = 0x00000004,
    VK_PIPELINE_CREATION_FEEDBACK_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkPipelineCreationFeedbackFlagBitsEXT;

```

- **VK\_PIPELINE\_CREATION\_FEEDBACK\_VALID\_BIT\_EXT** indicates that the feedback information is valid.
- **VK\_PIPELINE\_CREATION\_FEEDBACK\_APPLICATION\_PIPELINE\_CACHE\_HIT\_BIT\_EXT** indicates that a readily usable pipeline or pipeline stage was found in the `pipelineCache` specified by the application in the pipeline creation command.

An implementation **should** set the **VK\_PIPELINE\_CREATION\_FEEDBACK\_APPLICATION\_PIPELINE\_CACHE\_HIT\_BIT\_EXT** bit if it was able to avoid the large majority of pipeline or pipeline stage creation work by using the `pipelineCache` parameter of `vkCreateGraphicsPipelines`, `vkCreateRayTracingPipelinesNV`, or `vkCreateComputePipelines`. When an implementation sets this bit for the entire pipeline, it **may** leave it unset for any stage.

*Note*



Implementations are encouraged to provide a meaningful signal to applications using this bit. The intention is to communicate to the application that the pipeline or pipeline stage was created "as fast as it gets" using the pipeline cache provided by the application. If an implementation uses an internal cache, it is discouraged from setting this bit as the feedback would be unactionable.

- **VK\_PIPELINE\_CREATION\_FEEDBACK\_BASE\_PIPELINE\_ACCELERATION\_BIT\_EXT** indicates that the base pipeline specified by the `basePipelineHandle` or `basePipelineIndex` member of the `VkPipelineCreateInfo` structure was used to accelerate the creation of the pipeline.

An implementation **should** set the **VK\_PIPELINE\_CREATION\_FEEDBACK\_BASE\_PIPELINE\_ACCELERATION\_BIT\_EXT** bit if it was able to avoid a significant amount of work by using the base pipeline.

*Note*



While "significant amount of work" is subjective, implementations are encouraged to provide a meaningful signal to applications using this bit. For example, a 1% reduction in duration may not warrant setting this bit, while a 50% reduction would.

```
typedef VkFlags VkPipelineCreationFeedbackFlagsEXT;
```

`VkPipelineCreationFeedbackFlagsEXT` is a bitmask type for providing zero or more `VkPipelineCreationFeedbackFlagBitsEXT`.

# Chapter 10. Memory Allocation

Vulkan memory is broken up into two categories, *host memory* and *device memory*.

## 10.1. Host Memory

Host memory is memory needed by the Vulkan implementation for non-device-visible storage.

*Note*



This memory **may** be used to store the implementation's representation and state of Vulkan objects.

Vulkan provides applications the opportunity to perform host memory allocations on behalf of the Vulkan implementation. If this feature is not used, the implementation will perform its own memory allocations. Since most memory allocations are off the critical path, this is not meant as a performance feature. Rather, this **can** be useful for certain embedded systems, for debugging purposes (e.g. putting a guard page after all host allocations), or for memory allocation logging.

Allocators are provided by the application as a pointer to a `VkAllocationCallbacks` structure:

```
typedef struct VkAllocationCallbacks {
    void* pUserData;
    PFN_vkAllocationFunction pfnAllocation;
    PFN_vkReallocationFunction pfnReallocation;
    PFN_vkFreeFunction pfnFree;
    PFN_vkInternalAllocationNotification pfnInternalAllocation;
    PFN_vkInternalFreeNotification pfnInternalFree;
} VkAllocationCallbacks;
```

- `pUserData` is a value to be interpreted by the implementation of the callbacks. When any of the callbacks in `VkAllocationCallbacks` are called, the Vulkan implementation will pass this value as the first parameter to the callback. This value **can** vary each time an allocator is passed into a command, even when the same object takes an allocator in multiple commands.
- `pfnAllocation` is a `PFN_vkAllocationFunction` pointer to an application-defined memory allocation function.
- `pfnReallocation` is a `PFN_vkReallocationFunction` pointer to an application-defined memory reallocation function.
- `pfnFree` is a `PFN_vkFreeFunction` pointer to an application-defined memory free function.
- `pfnInternalAllocation` is a `PFN_vkInternalAllocationNotification` pointer to an application-defined function that is called by the implementation when the implementation makes internal allocations.
- `pfnInternalFree` is a `PFN_vkInternalFreeNotification` pointer to an application-defined function that is called by the implementation when the implementation frees internal allocations.

## Valid Usage

- `pfnAllocation` **must** be a valid pointer to a valid user-defined `PFN_vkAllocationFunction`
- `pfnReallocation` **must** be a valid pointer to a valid user-defined `PFN_vkReallocationFunction`
- `pfnFree` **must** be a valid pointer to a valid user-defined `PFN_vkFreeFunction`
- If either of `pfnInternalAllocation` or `pfnInternalFree` is not `NULL`, both **must** be valid callbacks

The type of `pfnAllocation` is:

```
typedef void* (VKAPI_PTR *PFN_vkAllocationFunction)(  
    void*  
        pUserData,  
    size_t  
        size,  
    size_t  
        alignment,  
    VkSystemAllocationScope  
        allocationScope);
```

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `size` is the size in bytes of the requested allocation.
- `alignment` is the requested alignment of the allocation in bytes and **must** be a power of two.
- `allocationScope` is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described [here](#).

If `pfnAllocation` is unable to allocate the requested memory, it **must** return `NULL`. If the allocation was successful, it **must** return a valid pointer to memory allocation containing at least `size` bytes, and with the pointer value being a multiple of `alignment`.

### Note

Correct Vulkan operation **cannot** be assumed if the application does not follow these rules.



For example, `pfnAllocation` (or `pfnReallocation`) could cause termination of running Vulkan instance(s) on a failed allocation for debugging purposes, either directly or indirectly. In these circumstances, it **cannot** be assumed that any part of any affected `VkInstance` objects are going to operate correctly (even `vkDestroyInstance`), and the application **must** ensure it cleans up properly via other means (e.g. process termination).

If `pfnAllocation` returns `NULL`, and if the implementation is unable to continue correct processing of the current command without the requested allocation, it **must** treat this as a run-time error, and generate `VK_ERROR_OUT_OF_HOST_MEMORY` at the appropriate time for the command in which the condition was detected, as described in [Return Codes](#).

If the implementation is able to continue correct processing of the current command without the requested allocation, then it **may** do so, and **must** not generate `VK_ERROR_OUT_OF_HOST_MEMORY` as a result of this failed allocation.

The type of `pfnReallocation` is:

```
typedef void* (VKAPI_PTR *PFN_vkReallocationFunction)(  
    void* pUserData,  
    void* pOriginal,  
    size_t size,  
    size_t alignment,  
    VkSystemAllocationScope allocationScope);
```

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `pOriginal` **must** be either `NULL` or a pointer previously returned by `pfnReallocation` or `pfnAllocation` of a compatible allocator.
- `size` is the size in bytes of the requested allocation.
- `alignment` is the requested alignment of the allocation in bytes and **must** be a power of two.
- `allocationScope` is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described [here](#).

`pfnReallocation` **must** return an allocation with enough space for `size` bytes, and the contents of the original allocation from bytes zero to `min(original size, new size) - 1` **must** be preserved in the returned allocation. If `size` is larger than the old size, the contents of the additional space are undefined. If satisfying these requirements involves creating a new allocation, then the old allocation **should** be freed.

If `pOriginal` is `NULL`, then `pfnReallocation` **must** behave equivalently to a call to `PFN_vkAllocationFunction` with the same parameter values (without `pOriginal`).

If `size` is zero, then `pfnReallocation` **must** behave equivalently to a call to `PFN_vkFreeFunction` with the same `pUserData` parameter value, and `pMemory` equal to `pOriginal`.

If `pOriginal` is non-`NULL`, the implementation **must** ensure that `alignment` is equal to the `alignment` used to originally allocate `pOriginal`.

If this function fails and `pOriginal` is non-`NULL` the application **must** not free the old allocation.

`pfnReallocation` **must** follow the same [rules for return values as PFN\\_vkAllocationFunction](#).

The type of `pfnFree` is:

```
typedef void (VKAPI_PTR *PFN_vkFreeFunction)(  
    void* pUserData,  
    void* pMemory);
```

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `pMemory` is the allocation to be freed.

`pMemory` may be `NULL`, which the callback **must** handle safely. If `pMemory` is non-`NULL`, it **must** be a pointer previously allocated by `pfnAllocation` or `pfnReallocation`. The application **should** free this memory.

The type of `pfnInternalAllocation` is:

```
typedef void (VKAPI_PTR *PFN_vkInternalAllocationNotification)(
    void* pUserData,
    size_t size,
    VkInternalAllocationType allocationType,
    VkSystemAllocationScope allocationScope);
```

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `size` is the requested size of an allocation.
- `allocationType` is a `VkInternalAllocationType` value specifying the requested type of an allocation.
- `allocationScope` is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described [here](#).

This is a purely informational callback.

The type of `pfnInternalFree` is:

```
typedef void (VKAPI_PTR *PFN_vkInternalFreeNotification)(
    void* pUserData,
    size_t size,
    VkInternalAllocationType allocationType,
    VkSystemAllocationScope allocationScope);
```

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `size` is the requested size of an allocation.
- `allocationType` is a `VkInternalAllocationType` value specifying the requested type of an allocation.
- `allocationScope` is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described [here](#).

Each allocation has an *allocation scope* defining its lifetime and which object it is associated with. Possible values passed to the `allocationScope` parameter of the callback functions specified by `VkAllocationCallbacks`, indicating the allocation scope, are:

```

typedef enum VkSystemAllocationScope {
    VK_SYSTEM_ALLOCATION_SCOPE_COMMAND = 0,
    VK_SYSTEM_ALLOCATION_SCOPE_OBJECT = 1,
    VK_SYSTEM_ALLOCATION_SCOPE_CACHE = 2,
    VK_SYSTEM_ALLOCATION_SCOPE_DEVICE = 3,
    VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE = 4,
    VK_SYSTEM_ALLOCATION_SCOPE_MAX_ENUM = 0x7FFFFFFF
} VkSystemAllocationScope;

```

- **VK\_SYSTEM\_ALLOCATION\_SCOPE\_COMMAND** specifies that the allocation is scoped to the duration of the Vulkan command.
- **VK\_SYSTEM\_ALLOCATION\_SCOPE\_OBJECT** specifies that the allocation is scoped to the lifetime of the Vulkan object that is being created or used.
- **VK\_SYSTEM\_ALLOCATION\_SCOPE\_CACHE** specifies that the allocation is scoped to the lifetime of a **VkPipelineCache** or **VkValidationCacheEXT** object.
- **VK\_SYSTEM\_ALLOCATION\_SCOPE\_DEVICE** specifies that the allocation is scoped to the lifetime of the Vulkan device.
- **VK\_SYSTEM\_ALLOCATION\_SCOPE\_INSTANCE** specifies that the allocation is scoped to the lifetime of the Vulkan instance.

Most Vulkan commands operate on a single object, or there is a sole object that is being created or manipulated. When an allocation uses an allocation scope of **VK\_SYSTEM\_ALLOCATION\_SCOPE\_OBJECT** or **VK\_SYSTEM\_ALLOCATION\_SCOPE\_CACHE**, the allocation is scoped to the object being created or manipulated.

When an implementation requires host memory, it will make callbacks to the application using the most specific allocator and allocation scope available:

- If an allocation is scoped to the duration of a command, the allocator will use the **VK\_SYSTEM\_ALLOCATION\_SCOPE\_COMMAND** allocation scope. The most specific allocator available is used: if the object being created or manipulated has an allocator, that object's allocator will be used, else if the parent **VkDevice** has an allocator it will be used, else if the parent **VkInstance** has an allocator it will be used. Else,
- If an allocation is associated with a **VkValidationCacheEXT** or **VkPipelineCache** object, the allocator will use the **VK\_SYSTEM\_ALLOCATION\_SCOPE\_CACHE** allocation scope. The most specific allocator available is used (cache, else device, else instance). Else,
- If an allocation is scoped to the lifetime of an object, that object is being created or manipulated by the command, and that object's type is not **VkDevice** or **VkInstance**, the allocator will use an allocation scope of **VK\_SYSTEM\_ALLOCATION\_SCOPE\_OBJECT**. The most specific allocator available is used (object, else device, else instance). Else,
- If an allocation is scoped to the lifetime of a device, the allocator will use an allocation scope of **VK\_SYSTEM\_ALLOCATION\_SCOPE\_DEVICE**. The most specific allocator available is used (device, else instance). Else,
- If the allocation is scoped to the lifetime of an instance and the instance has an allocator, its allocator will be used with an allocation scope of **VK\_SYSTEM\_ALLOCATION\_SCOPE\_INSTANCE**.

- Otherwise an implementation will allocate memory through an alternative mechanism that is unspecified.

Objects that are allocated from pools do not specify their own allocator. When an implementation requires host memory for such an object, that memory is sourced from the object's parent pool's allocator.

The application is not expected to handle allocating memory that is intended for execution by the host due to the complexities of differing security implementations across multiple platforms. The implementation will allocate such memory internally and invoke an application provided informational callback when these *internal allocations* are allocated and freed. Upon allocation of executable memory, `pfnInternalAllocation` will be called. Upon freeing executable memory, `pfnInternalFree` will be called. An implementation will only call an informational callback for executable memory allocations and frees.

The `allocationType` parameter to the `pfnInternalAllocation` and `pfnInternalFree` functions **may** be one of the following values:

```
typedef enum VkInternalAllocationType {
    VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE = 0,
    VK_INTERNAL_ALLOCATION_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkInternalAllocationType;
```

- `VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE` specifies that the allocation is intended for execution by the host.

An implementation **must** only make calls into an application-provided allocator during the execution of an API command. An implementation **must** only make calls into an application-provided allocator from the same thread that called the provoking API command. The implementation **should** not synchronize calls to any of the callbacks. If synchronization is needed, the callbacks **must** provide it themselves. The informational callbacks are subject to the same restrictions as the allocation callbacks.

If an implementation intends to make calls through a `VkAllocationCallbacks` structure between the time a `vkCreate*` command returns and the time a corresponding `vkDestroy*` command begins, that implementation **must** save a copy of the allocator before the `vkCreate*` command returns. The callback functions and any data structures they rely upon **must** remain valid for the lifetime of the object they are associated with.

If an allocator is provided to a `vkCreate*` command, a *compatible* allocator **must** be provided to the corresponding `vkDestroy*` command. Two `VkAllocationCallbacks` structures are compatible if memory allocated with `pfnAllocation` or `pfnReallocation` in each **can** be freed with `pfnReallocation` or `pfnFree` in the other. An allocator **must** not be provided to a `vkDestroy*` command if an allocator was not provided to the corresponding `vkCreate*` command.

If a non-`NULL` allocator is used, the `pfnAllocation`, `pfnReallocation` and `pfnFree` members **must** be non-`NULL` and point to valid implementations of the callbacks. An application **can** choose to not provide informational callbacks by setting both `pfnInternalAllocation` and `pfnInternalFree` to `NULL`. `pfnInternalAllocation` and `pfnInternalFree` **must** either both be `NULL` or both be non-`NULL`.

If `pfnAllocation` or `pfnReallocation` fail, the implementation **may** fail object creation and/or generate an `VK_ERROR_OUT_OF_HOST_MEMORY` error, as appropriate.

Allocation callbacks **must** not call any Vulkan commands.

The following sets of rules define when an implementation is permitted to call the allocator callbacks.

`pfnAllocation` or `pfnReallocation` **may** be called in the following situations:

- Allocations scoped to a `VkDevice` or `VkInstance` **may** be allocated from any API command.
- Allocations scoped to a command **may** be allocated from any API command.
- Allocations scoped to a `VkPipelineCache` **may** only be allocated from:
  - `vkCreatePipelineCache`
  - `vkMergePipelineCaches` for `dstCache`
  - `vkCreateGraphicsPipelines` for `pipelineCache`
  - `vkCreateComputePipelines` for `pipelineCache`
- Allocations scoped to a `VkValidationCacheEXT` **may** only be allocated from:
  - `vkCreateValidationCacheEXT`
  - `vkMergeValidationCachesEXT` for `dstCache`
  - `vkCreateShaderModule` for `validationCache` in `VkShaderModuleValidationCacheCreateInfoEXT`
- Allocations scoped to a `VkDescriptorPool` **may** only be allocated from:
  - any command that takes the pool as a direct argument
  - `vkAllocateDescriptorSets` for the `descriptorPool` member of its `pAllocateInfo` parameter
  - `vkCreateDescriptorPool`
- Allocations scoped to a `VkCommandPool` **may** only be allocated from:
  - any command that takes the pool as a direct argument
  - `vkCreateCommandPool`
  - `vkAllocateCommandBuffers` for the `commandPool` member of its `pAllocateInfo` parameter
  - any `vkCmd*` command whose `commandBuffer` was allocated from that `VkCommandPool`
- Allocations scoped to any other object **may** only be allocated in that object's `vkCreate*` command.

`pfnFree`, or `pfnReallocation` with zero `size`, **may** be called in the following situations:

- Allocations scoped to a `VkDevice` or `VkInstance` **may** be freed from any API command.
- Allocations scoped to a command **must** be freed by any API command which allocates such memory.
- Allocations scoped to a `VkPipelineCache` **may** be freed from `vkDestroyPipelineCache`.
- Allocations scoped to a `VkValidationCacheEXT` **may** be freed from `vkDestroyValidationCacheEXT`.
- Allocations scoped to a `VkDescriptorPool` **may** be freed from

- any command that takes the pool as a direct argument
- Allocations scoped to a `VkCommandPool` **may** be freed from:
  - any command that takes the pool as a direct argument
  - `vkResetCommandBuffer` whose `commandBuffer` was allocated from that `VkCommandPool`
- Allocations scoped to any other object **may** be freed in that object's `vkDestroy*` command.
- Any command that allocates host memory **may** also free host memory of the same scope.

## 10.2. Device Memory

*Device memory* is memory that is visible to the device — for example the contents of the image or buffer objects, which **can** be natively used by the device.

Memory properties of a physical device describe the memory heaps and memory types available.

To query memory properties, call:

```
void vkGetPhysicalDeviceMemoryProperties(
    VkPhysicalDevice                  physicalDevice,
    VkPhysicalDeviceMemoryProperties* pMemoryProperties);
```

- `physicalDevice` is the handle to the device to query.
- `pMemoryProperties` is a pointer to a `VkPhysicalDeviceMemoryProperties` structure in which the properties are returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pMemoryProperties` **must** be a valid pointer to a `VkPhysicalDeviceMemoryProperties` structure

The `VkPhysicalDeviceMemoryProperties` structure is defined as:

```
typedef struct VkPhysicalDeviceMemoryProperties {
    uint32_t      memoryTypeCount;
    VkMemoryType   memoryTypes[VK_MAX_MEMORY_TYPES];
    uint32_t      memoryHeapCount;
    VkMemoryHeap   memoryHeaps[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryProperties;
```

- `memoryTypeCount` is the number of valid elements in the `memoryTypes` array.
- `memoryTypes` is an array of `VK_MAX_MEMORY_TYPES` `VkMemoryType` structures describing the *memory types* that **can** be used to access memory allocated from the heaps specified by `memoryHeaps`.

- `memoryHeapCount` is the number of valid elements in the `memoryHeaps` array.
- `memoryHeaps` is an array of `VK_MAX_MEMORY_HEAPS` `VkMemoryHeap` structures describing the *memory heaps* from which memory **can** be allocated.

The `VkPhysicalDeviceMemoryProperties` structure describes a number of *memory heaps* as well as a number of *memory types* that **can** be used to access memory allocated in those heaps. Each heap describes a memory resource of a particular size, and each memory type describes a set of memory properties (e.g. host cached vs uncached) that **can** be used with a given memory heap. Allocations using a particular memory type will consume resources from the heap indicated by that memory type's heap index. More than one memory type **may** share each heap, and the heaps and memory types provide a mechanism to advertise an accurate size of the physical memory resources while allowing the memory to be used with a variety of different properties.

The number of memory heaps is given by `memoryHeapCount` and is less than or equal to `VK_MAX_MEMORY_HEAPS`. Each heap is described by an element of the `memoryHeaps` array as a `VkMemoryHeap` structure. The number of memory types available across all memory heaps is given by `memoryTypeCount` and is less than or equal to `VK_MAX_MEMORY_TYPES`. Each memory type is described by an element of the `memoryTypes` array as a `VkMemoryType` structure.

At least one heap **must** include `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT` in `VkMemoryHeap::flags`. If there are multiple heaps that all have similar performance characteristics, they **may** all include `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT`. In a unified memory architecture (UMA) system there is often only a single memory heap which is considered to be equally “local” to the host and to the device, and such an implementation **must** advertise the heap as device-local.

Each memory type returned by `vkGetPhysicalDeviceMemoryProperties` **must** have its `propertyFlags` set to one of the following values:

- 0
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT`
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`

- VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT
- VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT |  
VK\_MEMORY\_PROPERTY\_LAZILY\_ALLOCATED\_BIT
- VK\_MEMORY\_PROPERTY\_PROTECTED\_BIT
- VK\_MEMORY\_PROPERTY\_PROTECTED\_BIT | VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT
- VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_CACHED\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_CACHED\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD |  
VK\_MEMORY\_PROPERTY\_DEVICE\_UNCACHED\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_CACHED\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD |  
VK\_MEMORY\_PROPERTY\_DEVICE\_UNCACHED\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD |  
VK\_MEMORY\_PROPERTY\_DEVICE\_UNCACHED\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT |  
VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD |  
VK\_MEMORY\_PROPERTY\_DEVICE\_UNCACHED\_BIT\_AMD
- VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  
VK\_MEMORY\_PROPERTY\_HOST\_CACHED\_BIT |

```
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT |  
VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD |  
VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD
```

There **must** be at least one memory type with both the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` bits set in its `propertyFlags`. There **must** be at least one memory type with the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` bit set in its `propertyFlags`. If the `deviceCoherentMemory` feature is enabled, there **must** be at least one memory type with the `VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD` bit set in its `propertyFlags`.

For each pair of elements **X** and **Y** returned in `memoryTypes`, **X** **must** be placed at a lower index position than **Y** if:

- either the set of bit flags returned in the `propertyFlags` member of **X** is a strict subset of the set of bit flags returned in the `propertyFlags` member of **Y**; or
- the `propertyFlags` members of **X** and **Y** are equal, and **X** belongs to a memory heap with greater performance (as determined in an implementation-specific manner); or
- or the `propertyFlags` members of **X** includes `VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD` or `VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD` and **Y** does not

*Note*

There is no ordering requirement between **X** and **Y** elements for the case their `propertyFlags` members are not in a subset relation. That potentially allows more than one possible way to order the same set of memory types. Notice that the [list of all allowed memory property flag combinations](#) is written in a valid order. But if instead `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` was before `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`, the list would still be in a valid order.



There may be a performance penalty for using device coherent or uncached device memory types, and using these accidentally is undesirable. In order to avoid this, memory types with these properties always appear at the end of the list; but are subject to the same rules otherwise.

This ordering requirement enables applications to use a simple search loop to select the desired memory type along the lines of:

```

// Find a memory in `memoryTypeBitsRequirement` that includes all of
// `requiredProperties`
int32_t findProperties(const VkPhysicalDeviceMemoryProperties* pMemoryProperties,
                      uint32_t memoryTypeBitsRequirement,
                      VkMemoryPropertyFlags requiredProperties) {
    const uint32_t memoryCount = pMemoryProperties->memoryTypeCount;
    for (uint32_t memoryIndex = 0; memoryIndex < memoryCount; ++memoryIndex) {
        const uint32_t memoryTypeBits = (1 << memoryIndex);
        const bool isRequiredMemoryType = memoryTypeBitsRequirement & memoryTypeBits;

        const VkMemoryPropertyFlags properties =
            pMemoryProperties->memoryTypes[memoryIndex].propertyFlags;
        const bool hasRequiredProperties =
            (properties & requiredProperties) == requiredProperties;

        if (isRequiredMemoryType && hasRequiredProperties)
            return static_cast<int32_t>(memoryIndex);
    }

    // failed to find memory type
    return -1;
}

// Try to find an optimal memory type, or if it does not exist try fallback memory
// type
// 'device' is the VkDevice
// 'image' is the VkImage that requires memory to be bound
// 'memoryProperties' properties as returned by vkGetPhysicalDeviceMemoryProperties
// 'requiredProperties' are the property flags that must be present
// 'optimalProperties' are the property flags that are preferred by the application
VkMemoryRequirements memoryRequirements;
vkGetImageMemoryRequirements(device, image, &memoryRequirements);
int32_t memoryType =
    findProperties(&memoryProperties, memoryRequirements.memoryTypeBits,
optimalProperties);
if (memoryType == -1) // not found; try fallback properties
    memoryType =
        findProperties(&memoryProperties, memoryRequirements.memoryTypeBits,
requiredProperties);

```

To query memory properties, call:

```

void vkGetPhysicalDeviceMemoryProperties2(
    VkPhysicalDevice                                     physicalDevice,
    VkPhysicalDeviceMemoryProperties2*                  pMemoryProperties);

```

or the equivalent command

```
void vkGetPhysicalDeviceMemoryProperties2KHR(  
    VkPhysicalDevice physicalDevice,  
    VkPhysicalDeviceMemoryProperties2* pMemoryProperties);
```

- `physicalDevice` is the handle to the device to query.
- `pMemoryProperties` is a pointer to a `VkPhysicalDeviceMemoryProperties2` structure in which the properties are returned.

`vkGetPhysicalDeviceMemoryProperties2` behaves similarly to `vkGetPhysicalDeviceMemoryProperties`, with the ability to return extended information in a `pNext` chain of output structures.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pMemoryProperties` **must** be a valid pointer to a `VkPhysicalDeviceMemoryProperties2` structure

The `VkPhysicalDeviceMemoryProperties2` structure is defined as:

```
typedef struct VkPhysicalDeviceMemoryProperties2 {  
    VkStructureType sType;  
    void* pNext;  
    VkPhysicalDeviceMemoryProperties memoryProperties;  
} VkPhysicalDeviceMemoryProperties2;
```

or the equivalent

```
typedef VkPhysicalDeviceMemoryProperties2 VkPhysicalDeviceMemoryProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memoryProperties` is a `VkPhysicalDeviceMemoryProperties` structure which is populated with the same values as in `vkGetPhysicalDeviceMemoryProperties`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkPhysicalDeviceMemoryBudgetPropertiesEXT`

The `VkMemoryHeap` structure is defined as:

```
typedef struct VkMemoryHeap {
    VkDeviceSize      size;
    VkMemoryHeapFlags flags;
} VkMemoryHeap;
```

- `size` is the total memory size in bytes in the heap.
- `flags` is a bitmask of `VkMemoryHeapFlagBits` specifying attribute flags for the heap.

Bits which **may** be set in `VkMemoryHeap::flags`, indicating attribute flags for the heap, are:

```
typedef enum VkMemoryHeapFlagBits {
    VK_MEMORY_HEAP_DEVICE_LOCAL_BIT = 0x00000001,
    VK_MEMORY_HEAP_MULTI_INSTANCE_BIT = 0x00000002,
    VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHR = VK_MEMORY_HEAP_MULTI_INSTANCE_BIT,
    VK_MEMORY_HEAP_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkMemoryHeapFlagBits;
```

- `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT` specifies that the heap corresponds to device local memory. Device local memory **may** have different performance characteristics than host local memory, and **may** support different memory property flags.
- `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` specifies that in a logical device representing more than one physical device, there is a per-physical device instance of the heap memory. By default, an allocation from such a heap will be replicated to each physical device's instance of the heap.

```
typedef VkFlags VkMemoryHeapFlags;
```

`VkMemoryHeapFlags` is a bitmask type for setting a mask of zero or more `VkMemoryHeapFlagBits`.

The `VkMemoryType` structure is defined as:

```
typedef struct VkMemoryType {
    VkMemoryPropertyFlags   propertyFlags;
    uint32_t                heapIndex;
} VkMemoryType;
```

- `heapIndex` describes which memory heap this memory type corresponds to, and **must** be less than `memoryHeapCount` from the `VkPhysicalDeviceMemoryProperties` structure.
- `propertyFlags` is a bitmask of `VkMemoryPropertyFlagBits` of properties for this memory type.

Bits which **may** be set in `VkMemoryType::propertyFlags`, indicating properties of a memory heap, are:

```

typedef enum VkMemoryPropertyFlagBits {
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT = 0x00000001,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT = 0x00000002,
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004,
    VK_MEMORY_PROPERTY_HOST_CACHED_BIT = 0x00000008,
    VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT = 0x00000010,
    VK_MEMORY_PROPERTY_PROTECTED_BIT = 0x00000020,
    VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD = 0x00000040,
    VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD = 0x00000080,
    VK_MEMORY_PROPERTY_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkMemoryPropertyFlagBits;

```

- **VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT** bit specifies that memory allocated with this type is the most efficient for device access. This property will be set if and only if the memory type belongs to a heap with the **VK\_MEMORY\_HEAP\_DEVICE\_LOCAL\_BIT** set.
- **VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT** bit specifies that memory allocated with this type **can** be mapped for host access using [vkMapMemory](#).
- **VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT** bit specifies that the host cache management commands [vkFlushMappedMemoryRanges](#) and [vkInvalidateMappedMemoryRanges](#) are not needed to flush host writes to the device or make device writes visible to the host, respectively.
- **VK\_MEMORY\_PROPERTY\_HOST\_CACHED\_BIT** bit specifies that memory allocated with this type is cached on the host. Host memory accesses to uncached memory are slower than to cached memory, however uncached memory is always host coherent.
- **VK\_MEMORY\_PROPERTY\_LAZILY\_ALLOCATED\_BIT** bit specifies that the memory type only allows device access to the memory. Memory types **must** not have both **VK\_MEMORY\_PROPERTY\_LAZILY\_ALLOCATED\_BIT** and **VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT** set. Additionally, the object's backing memory **may** be provided by the implementation lazily as specified in [Lazily Allocated Memory](#).
- **VK\_MEMORY\_PROPERTY\_PROTECTED\_BIT** bit specifies that the memory type only allows device access to the memory, and allows protected queue operations to access the memory. Memory types **must** not have **VK\_MEMORY\_PROPERTY\_PROTECTED\_BIT** set and any of **VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT** set, or **VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT** set, or **VK\_MEMORY\_PROPERTY\_HOST\_CACHED\_BIT** set.
- **VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD** bit specifies that device accesses to allocations of this memory type are automatically made available and visible.
- **VK\_MEMORY\_PROPERTY\_DEVICE\_UNCACHED\_BIT\_AMD** bit specifies that memory allocated with this type is not cached on the device. Uncached device memory is always device coherent.

For any memory allocated with both the **VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT** and the **VK\_MEMORY\_PROPERTY\_DEVICE\_COHERENT\_BIT\_AMD**, host or device accesses also perform automatic memory domain transfer operations, such that writes are always automatically available and visible to both host and device memory domains.

*Note*

Device coherence is a useful property for certain debugging use cases (e.g. crash analysis, where performing separate coherence actions could mean values are not reported correctly). However, device coherent accesses may be slower than equivalent accesses without device coherence, particularly if they are also device uncached. For device uncached memory in particular, repeated accesses to the same or neighbouring memory locations over a short time period (e.g. within a frame) may be slower than it would be for the equivalent cached memory type. As such, it's generally inadvisable to use device coherent or device uncached memory except when really needed.

```
typedef VkFlags VkMemoryPropertyFlags;
```

`VkMemoryPropertyFlags` is a bitmask type for setting a mask of zero or more `VkMemoryPropertyFlagBits`.

If the `VkPhysicalDeviceMemoryBudgetPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceMemoryProperties2`, it is filled with the current memory budgets and usages.

The `VkPhysicalDeviceMemoryBudgetPropertiesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceMemoryBudgetPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkDeviceSize       heapBudget[VK_MAX_MEMORY_HEAPS];
    VkDeviceSize       heapUsage[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryBudgetPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `heapBudget` is an array of `VK_MAX_MEMORY_HEAPS` `VkDeviceSize` values in which memory budgets are returned, with one element for each memory heap. A heap's budget is a rough estimate of how much memory the process **can** allocate from that heap before allocations **may** fail or cause performance degradation. The budget includes any currently allocated device memory.
- `heapUsage` is an array of `VK_MAX_MEMORY_HEAPS` `VkDeviceSize` values in which memory usages are returned, with one element for each memory heap. A heap's usage is an estimate of how much memory the process is currently using in that heap.

The values returned in this structure are not invariant. The `heapBudget` and `heapUsage` values **must** be zero for array elements greater than or equal to `VkPhysicalDeviceMemoryProperties::memoryHeapCount`. The `heapBudget` value **must** be non-zero for array elements less than `VkPhysicalDeviceMemoryProperties::memoryHeapCount`. The `heapBudget` value **must** be less than or equal to `VkMemoryHeap::size` for each heap.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT`

A Vulkan device operates on data in device memory via memory objects that are represented in the API by a `VkDeviceMemory` handle:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDeviceMemory)
```

To allocate memory objects, call:

```
VkResult vkAllocateMemory(  
    VkDevice device,  
    const VkMemoryAllocateInfo* pAllocateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkDeviceMemory* pMemory);
```

- `device` is the logical device that owns the memory.
- `pAllocateInfo` is a pointer to a `VkMemoryAllocateInfo` structure describing parameters of the allocation. A successful returned allocation **must** use the requested parameters—no substitution is permitted by the implementation.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pMemory` is a pointer to a `VkDeviceMemory` handle in which information about the allocated memory is returned.

Allocations returned by `vkAllocateMemory` are guaranteed to meet any alignment requirement of the implementation. For example, if an implementation requires 128 byte alignment for images and 64 byte alignment for buffers, the device memory returned through this mechanism would be 128-byte aligned. This ensures that applications **can** correctly suballocate objects of different types (with potentially different alignment requirements) in the same memory object.

When memory is allocated, its contents are undefined with the following constraint:

- The contents of unprotected memory **must** not be a function of data protected memory objects, even if those memory objects were previously freed.

### Note



The contents of memory allocated by one application **should** not be a function of data from protected memory objects of another application, even if those memory objects were previously freed.

The maximum number of valid memory allocations that **can** exist simultaneously within a `VkDevice` **may** be restricted by implementation- or platform-dependent limits. If a call to `vkAllocateMemory` would cause the total number of allocations to exceed these limits, such a call

will fail and **must** return `VK_ERROR_TOO_MANY_OBJECTS`. The `maxMemoryAllocationCount` feature describes the number of allocations that **can** exist simultaneously before encountering these internal limits.

Some platforms **may** have a limit on the maximum size of a single allocation. For example, certain systems **may** fail to create allocations with a size greater than or equal to 4GB. Such a limit is implementation-dependent, and if such a failure occurs then the error `VK_ERROR_OUT_OF_DEVICE_MEMORY` **must** be returned. This limit is advertised in `VkPhysicalDeviceMaintenance3Properties::maxMemoryAllocationSize`.

The cumulative memory size allocated to a heap **can** be limited by the size of the specified heap. In such cases, allocated memory is tracked on a per-device and per-heap basis. Some platforms allow overallocation into other heaps. The overallocation behavior **can** be specified through the `VK_AMD_memory_overallocation_behavior` extension.

## Valid Usage

- `pAllocateInfo->allocationSize` **must** be less than or equal to `VkPhysicalDeviceMemoryProperties::memoryHeaps[memindex].size` where `memindex = VkPhysicalDeviceMemoryProperties::memoryTypes[pAllocateInfo->memoryTypeIndex].heapIndex` as returned by `vkGetPhysicalDeviceMemoryProperties` for the `VkPhysicalDevice` that `device` was created from.
- `pAllocateInfo->memoryTypeIndex` **must** be less than `VkPhysicalDeviceMemoryProperties ::memoryTypeCount` as returned by `vkGetPhysicalDeviceMemoryProperties` for the `VkPhysicalDevice` that `device` was created from.
- If the `deviceCoherentMemory` feature is not enabled, `pAllocateInfo->memoryTypeIndex` **must** not identify a memory type supporting `VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pAllocateInfo` **must** be a valid pointer to a valid `VkMemoryAllocateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pMemory` **must** be a valid pointer to a `VkDeviceMemory` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE`
- `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR`

The `VkMemoryAllocateInfo` structure is defined as:

```
typedef struct VkMemoryAllocateInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceSize       allocationSize;
    uint32_t           memoryTypeIndex;
} VkMemoryAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `allocationSize` is the size of the allocation in bytes
- `memoryTypeIndex` is an index identifying a memory type from the `memoryTypes` array of the `VkPhysicalDeviceMemoryProperties` structure

An instance of the `VkMemoryAllocateInfo` structure defines a memory import operation if the `pNext` chain contains an instance of one of the following structures:

- `VkImportMemoryWin32HandleInfoKHR` with non-zero `handleType` value
- `VkImportMemoryFdInfoKHR` with a non-zero `handleType` value
- `VkImportMemoryHostPointerInfoEXT` with a non-zero `handleType` value
- `VkImportAndroidHardwareBufferInfoANDROID` with a non-`NULL` `buffer` value

Importing memory **must** not modify the content of the memory. Implementations **must** ensure that importing memory does not enable the importing Vulkan instance to access any memory or resources in other Vulkan instances other than that corresponding to the memory object imported. Implementations **must** also ensure accessing imported memory which has not been initialized does not allow the importing Vulkan instance to obtain data from the exporting Vulkan instance or vice-versa.

*Note*



How exported and imported memory is isolated is left to the implementation, but applications should be aware that such isolation **may** prevent implementations from placing multiple exportable memory objects in the same physical or virtual page. Hence, applications **should** avoid creating many small external memory objects whenever possible.

When performing a memory import operation, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations **must** perform sufficient validation of external handles to ensure that the operation results in a valid memory object which will not cause program termination, device loss, queue stalls, or corruption of other resources when used as allowed according to its allocation parameters. If the external handle provided does not meet these requirements, the implementation **must** fail the memory import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

## Valid Usage

- If the `pNext` chain contains an instance of `VkExportMemoryAllocateInfo`, and any of the handle types specified in `VkExportMemoryAllocateInfo::handleTypes` require a dedicated allocation, as reported by `vkGetPhysicalDeviceImageFormatProperties2` in `VkExternalImageFormatProperties::externalMemoryProperties::externalMemoryFeatures` or `VkExternalBufferProperties::externalMemoryProperties::externalMemoryFeatures`, the `pNext` chain must contain an instance of `VkMemoryDedicatedAllocateInfo` or `VkDedicatedAllocationMemoryAllocateInfoNV` with either its `image` or `buffer` field set to a value other than `VK_NULL_HANDLE`.
- If the `pNext` chain contains an instance of `VkExportMemoryAllocateInfo`, it **must** not contain an instance of `VkExportMemoryAllocateInfoNV` or `VkExportMemoryWin32HandleInfoNV`.
- If the `pNext` chain contains an instance of `VkImportMemoryWin32HandleInfoKHR`, it **must** not contain an instance of `VkImportMemoryWin32HandleInfoNV`.
- If the parameters define an import operation, the external handle specified was created by the Vulkan API, and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHR`, then the values of `allocationSize` and `memoryTypeIndex` **must** match those specified when the memory object being imported was created.
- If the parameters define an import operation and the external handle specified was created by the Vulkan API, the device mask specified by `VkMemoryAllocateFlagsInfo` **must** match that specified when the memory object being imported was allocated.
- If the parameters define an import operation and the external handle specified was created by the Vulkan API, the list of physical devices that comprise the logical device passed to `vkAllocateMemory` **must** match the list of physical devices that comprise the logical device on which the memory was originally allocated.
- If the parameters define an import operation and the external handle is an NT handle or a global share handle created outside of the Vulkan API, the value of `memoryTypeIndex` **must** be one of those returned by `vkGetMemoryWin32HandlePropertiesKHR`.
- If the parameters define an import operation, the external handle was created by the Vulkan API, and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR`, then the values of `allocationSize` and `memoryTypeIndex` **must** match those specified when the memory object being imported was created.
- If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, `allocationSize` **must** match the size reported in the memory requirements of the `image` or `buffer` member of the instance of `VkDedicatedAllocationMemoryAllocateInfoNV` included in the `pNext` chain.
- If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, `allocationSize` **must** match the size

specified when creating the Direct3D 12 heap from which the external handle was extracted.

- If the parameters define an import operation and the external handle is a POSIX file descriptor created outside of the Vulkan API, the value of `memoryTypeIndex` **must** be one of those returned by [vkGetMemoryFdPropertiesKHR](#).
- If the protected memory feature is not enabled, the `VkMemoryAllocateInfo::memoryTypeIndex` **must** not indicate a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`.
- If the parameters define an import operation and the external handle is a host pointer, the value of `memoryTypeIndex` **must** be one of those returned by [vkGetMemoryHostPointerPropertiesEXT](#)
- If the parameters define an import operation and the external handle is a host pointer, `allocationSize` **must** be an integer multiple of [VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment](#)
- If the parameters define an import operation and the external handle is a host pointer, the `pNext` chain **must** not contain an instance of [VkDedicatedAllocationMemoryAllocateInfoNV](#) with either its `image` or `buffer` field set to a value other than `VK_NULL_HANDLE`.
- If the parameters define an import operation and the external handle is a host pointer, the `pNext` chain **must** not contain an instance of [VkMemoryDedicatedAllocateInfo](#) with either its `image` or `buffer` field set to a value other than `VK_NULL_HANDLE`.
- If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `allocationSize` **must** be the size returned by [vkGetAndroidHardwareBufferPropertiesANDROID](#) for the Android hardware buffer.
- If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, and the `pNext` chain does not contain an instance of [VkMemoryDedicatedAllocateInfo](#) or [VkMemoryDedicatedAllocateInfo::image](#) is `VK_NULL_HANDLE`, the Android hardware buffer **must** have a `AHardwareBuffer_Desc::format` of `AHARDWAREBUFFER_FORMAT_BLOB` and a `AHardwareBuffer_Desc::usage` that includes `AHARDWAREBUFFER_USAGE_GPU_DATA_BUFFER`.
- If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `memoryTypeIndex` **must** be one of those returned by [vkGetAndroidHardwareBufferPropertiesANDROID](#) for the Android hardware buffer.
- If the parameters do not define an import operation, and the `pNext` chain contains an instance of [VkExportMemoryAllocateInfo](#) with `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` included in its `handleTypes` member, and the `pNext` contains an instance of [VkMemoryDedicatedAllocateInfo](#) with `image` not equal to `VK_NULL_HANDLE`, then `allocationSize` **must** be `0`, otherwise `allocationSize` **must** be greater than `0`.
- If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes an instance of [VkMemoryDedicatedAllocateInfo](#) with `image` that is not `VK_NULL_HANDLE`, the Android hardware buffer's [AHardwareBuffer](#)

`::usage` **must** include at least one of `AHARDWAREBUFFER_USAGE_GPU_COLOR_OUTPUT` or `AHARDWAREBUFFER_USAGE_GPU_SAMPLED_IMAGE`.

- If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes an instance of `VkMemoryDedicatedAllocateInfo` with `image` that is not `VK_NULL_HANDLE`, the format of `image` **must** be `VK_FORMAT_UNDEFINED` or the format returned by `vkGetAndroidHardwareBufferPropertiesANDROID` in `VkAndroidHardwareBufferFormatPropertiesANDROID::format` for the Android hardware buffer.
- If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes an instance of `VkMemoryDedicatedAllocateInfo` with `image` that is not `VK_NULL_HANDLE`, the width, height, and array layer dimensions of `image` and the Android hardware buffer's `AHardwareBuffer_Desc` **must** be identical.
- If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes an instance of `VkMemoryDedicatedAllocateInfo` with `image` that is not `VK_NULL_HANDLE`, and the Android hardware buffer's `AHardwareBuffer::usage` includes `AHARDWAREBUFFER_USAGE_GPU_MIPMAP_COMPLETE`, the `image` **must** have a complete mipmap chain.
- If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes an instance of `VkMemoryDedicatedAllocateInfo` with `image` that is not `VK_NULL_HANDLE`, and the Android hardware buffer's `AHardwareBuffer::usage` does not include `AHARDWAREBUFFER_USAGE_GPU_MIPMAP_COMPLETE`, the `image` **must** have exactly one mipmap level.
- If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes an instance of `VkMemoryDedicatedAllocateInfo` with `image` that is not `VK_NULL_HANDLE`, each bit set in the usage of `image` **must** be listed in `AHardwareBuffer Usage Equivalence`, and if there is a corresponding `AHARDWAREBUFFER_USAGE` bit listed that bit **must** be included in the Android hardware buffer's `AHardwareBuffer_Desc::usage`.
- If `VkMemoryOpaqueCaptureAddressAllocateInfoKHR::opaqueCaptureAddress` is not zero, `VkMemoryAllocateFlagsInfo::flags` **must** include `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`
- If `VkMemoryAllocateFlagsInfo::flags` includes `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`, the `bufferDeviceAddressCaptureReplay` feature **must** be enabled
- If `VkMemoryAllocateFlagsInfo::flags` includes `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR`, the `bufferDeviceAddress` feature **must** be enabled
- If the `pNext` chain contains an instance of `VkImportMemoryHostPointerInfoEXT`, `VkMemoryOpaqueCaptureAddressAllocateInfoKHR::opaqueCaptureAddress` **must** be zero
- If the parameters define an import operation, `VkMemoryOpaqueCaptureAddressAllocateInfoKHR::opaqueCaptureAddress` **must** be zero

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDedicatedAllocationMemoryAllocateInfoNV`,  
`VkExportMemoryAllocateInfo`,  
`VkExportMemoryAllocateInfoNV`,  
`VkExportMemoryWin32HandleInfoKHR`,  
`VkExportMemoryWin32HandleInfoNV`,  
`VkImportAndroidHardwareBufferInfoANDROID`,  
`VkImportMemoryFdInfoKHR`,  
`VkImportMemoryHostPointerInfoEXT`,  
`VkImportMemoryWin32HandleInfoKHR`,  
`VkImportMemoryWin32HandleInfoNV`,  
`VkMemoryAllocateFlagsInfo`,  
`VkMemoryDedicatedAllocateInfo`, `VkMemoryOpaqueCaptureAddressAllocateInfoKHR`, or  
`VkMemoryPriorityAllocateInfoEXT`
- Each `sType` member in the `pNext` chain must be unique

If the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` structure, then that structure includes a handle of the sole buffer or image resource that the memory can be bound to.

The `VkMemoryDedicatedAllocateInfo` structure is defined as:

```
typedef struct VkMemoryDedicatedAllocateInfo {  
    VkStructureType    sType;  
    const void*        pNext;  
    VkImage            image;  
    VkBuffer           buffer;  
} VkMemoryDedicatedAllocateInfo;
```

or the equivalent

```
typedef VkMemoryDedicatedAllocateInfo VkMemoryDedicatedAllocateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `image` is `VK_NULL_HANDLE` or a handle of an image which this memory will be bound to.
- `buffer` is `VK_NULL_HANDLE` or a handle of a buffer which this memory will be bound to.

## Valid Usage

- At least one of `image` and `buffer` **must** be `VK_NULL_HANDLE`
- If `image` is not `VK_NULL_HANDLE`, `VkMemoryAllocateInfo::allocationSize` **must** equal the `VkMemoryRequirements::size` of the image
- If `image` is not `VK_NULL_HANDLE`, `image` **must** have been created without `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` set in `VkImageCreateInfo::flags`
- If `buffer` is not `VK_NULL_HANDLE`, `VkMemoryAllocateInfo::allocationSize` **must** equal the `VkMemoryRequirements::size` of the buffer
- If `buffer` is not `VK_NULL_HANDLE`, `buffer` **must** have been created without `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` set in `VkBufferCreateInfo::flags`
- If `image` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation with handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT`,  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`,  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`,  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, or  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, and the external handle was created by the Vulkan API, then the memory being imported **must** also be a dedicated image allocation and `image` must be identical to the image associated with the imported memory.
- If `buffer` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation with handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT`,  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`,  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`,  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, or  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, and the external handle was created by the Vulkan API, then the memory being imported **must** also be a dedicated buffer allocation and `buffer` must be identical to the buffer associated with the imported memory.
- If `image` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation with handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT`, the memory being imported **must** also be a dedicated image allocation and `image` must be identical to the image associated with the imported memory.
- If `buffer` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation with handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT`, the memory being imported **must** also be a dedicated buffer allocation and `buffer` must be identical to the buffer associated with the imported memory.
- If `image` is not `VK_NULL_HANDLE`, `image` **must** not have been created with `VK_IMAGE_CREATE_DISJOINT_BIT` set in `VkImageCreateInfo::flags`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO`
- If `image` is not `VK_NULL_HANDLE`, `image` must be a valid `VkImage` handle
- If `buffer` is not `VK_NULL_HANDLE`, `buffer` must be a valid `VkBuffer` handle
- Both of `buffer`, and `image` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

If the `pNext` chain includes a `VkDedicatedAllocationMemoryAllocateInfoNV` structure, then that structure includes a handle of the sole buffer or image resource that the memory can be bound to.

The `VkDedicatedAllocationMemoryAllocateInfoNV` structure is defined as:

```
typedef struct VkDedicatedAllocationMemoryAllocateInfoNV {
    VkStructureType    sType;
    const void*        pNext;
    VkImage            image;
    VkBuffer           buffer;
} VkDedicatedAllocationMemoryAllocateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `image` is `VK_NULL_HANDLE` or a handle of an image which this memory will be bound to.
- `buffer` is `VK_NULL_HANDLE` or a handle of a buffer which this memory will be bound to.

## Valid Usage

- At least one of `image` and `buffer` **must** be `VK_NULL_HANDLE`
- If `image` is not `VK_NULL_HANDLE`, the image **must** have been created with `VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`
- If `buffer` is not `VK_NULL_HANDLE`, the buffer **must** have been created with `VkDedicatedAllocationBufferCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`
- If `image` is not `VK_NULL_HANDLE`, `VkMemoryAllocateInfo::allocationSize` **must** equal the `VkMemoryRequirements::size` of the image
- If `buffer` is not `VK_NULL_HANDLE`, `VkMemoryAllocateInfo::allocationSize` **must** equal the `VkMemoryRequirements::size` of the buffer
- If `image` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation, the memory being imported **must** also be a dedicated image allocation and `image` **must** be identical to the image associated with the imported memory.
- If `buffer` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation, the memory being imported **must** also be a dedicated buffer allocation and `buffer` **must** be identical to the buffer associated with the imported memory.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPEDEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV`
- If `image` is not `VK_NULL_HANDLE`, `image` **must** be a valid `VkImage` handle
- If `buffer` is not `VK_NULL_HANDLE`, `buffer` **must** be a valid `VkBuffer` handle
- Both of `buffer`, and `image` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

If the `pNext` chain includes a `VkMemoryPriorityAllocateInfoEXT` structure, then that structure includes a priority for the memory.

The `VkMemoryPriorityAllocateInfoEXT` structure is defined as:

```
typedef struct VkMemoryPriorityAllocateInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    float              priority;
} VkMemoryPriorityAllocateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `priority` is a floating-point value between `0` and `1`, indicating the priority of the allocation relative to other memory allocations. Larger values are higher priority. The granularity of the

priorities is implementation-dependent.

Memory allocations with higher priority **may** be more likely to stay in device-local memory when the system is under memory pressure.

If this structure is not included, it is as if the `priority` value were `0.5`.

## Valid Usage

- `priority` **must** be between `0` and `1`, inclusive

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_PRIORITY_ALLOCATE_INFO_EXT`

When allocating memory that **may** be exported to another process or Vulkan instance, add a `VkExportMemoryAllocateInfo` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure, specifying the handle types that **may** be exported.

The `VkExportMemoryAllocateInfo` structure is defined as:

```
typedef struct VkExportMemoryAllocateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkExternalMemoryHandleTypeFlags handleTypes;
} VkExportMemoryAllocateInfo;
```

or the equivalent

```
typedef VkExportMemoryAllocateInfo VkExportMemoryAllocateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying one or more memory handle types the application **can** export from the resulting allocation. The application **can** request multiple handle types for the same allocation.

## Valid Usage

- The bits in `handleTypes` **must** be supported and compatible, as reported by `VkExternalImageFormatProperties` or `VkExternalBufferProperties`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO`
- `handleTypes` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values

To specify additional attributes of NT handles exported from a memory object, add the `VkExportMemoryWin32HandleInfoKHR` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure. The `VkExportMemoryWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkExportMemoryWin32HandleInfoKHR {
    VkStructureType          sType;
    const void*               pNext;
    const SECURITY_ATTRIBUTES* pAttributes;
    DWORD                    dwAccess;
    LPCWSTR                  name;
} VkExportMemoryWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pAttributes` is a pointer to a Windows `SECURITY_ATTRIBUTES` structure specifying security attributes of the handle.
- `dwAccess` is a `DWORD` specifying access rights of the handle.
- `name` is a null-terminated UTF-16 string to associate with the underlying resource referenced by NT handles exported from the created memory.

If this structure is not present, or if `pAttributes` is set to `NULL`, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in the MSDN documentation for “Synchronization Object Security and Access Rights”<sup>1</sup>. Further, if the structure is not present, the access rights used depend on the handle type.

For handles of the following types:

`VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`  
`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`

The implementation **must** ensure the access rights allow read and write access to the memory.

For handles of the following types:

`VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT` `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`

The access rights **must** be:

`GENERIC_ALL`

<https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-object-security-and-access-rights>

## Valid Usage

- If `VkExportMemoryAllocateInfo::handleTypes` does not include `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, `VkExportMemoryWin32HandleInfoKHR` **must** not be in the `pNext` chain of `VkMemoryAllocateInfo`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR`
- If `pAttributes` is not `NULL`, `pAttributes` **must** be a valid pointer to a valid `SECURITY_ATTRIBUTES` value

To import memory from a Windows handle, add a `VkImportMemoryWin32HandleInfoKHR` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure.

The `VkImportMemoryWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkImportMemoryWin32HandleInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    HANDLE                   handle;
    LPCWSTR                  name;
} VkImportMemoryWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleType` specifies the type of `handle` or `name`.
- `handle` is the external handle to import, or `NULL`.
- `name` is a null-terminated UTF-16 string naming the underlying memory resource to import, or `NULL`.

Importing memory objects from Windows handles does not transfer ownership of the handle to the Vulkan implementation. For handle types defined as NT handles, the application **must** release ownership using the `CloseHandle` system call when the handle is no longer needed.

Applications **can** import the same underlying memory into multiple instances of Vulkan, into the

same instance from which it was exported, and multiple times into a given Vulkan instance. In all cases, each import operation **must** create a distinct `VkDeviceMemory` object.

## Valid Usage

- If `handleType` is not `0`, it **must** be supported for import, as reported by `VkExternalImageFormatProperties` or `VkExternalBufferProperties`.
- The memory from which `handle` was exported, or the memory named by `name` **must** have been created on the same underlying physical device as `device`.
- If `handleType` is not `0`, it **must** be defined as an NT handle or a global share handle.
- If `handleType` is not `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, `name` **must** be `NULL`.
- If `handleType` is not `0` and `handle` is `NULL`, `name` **must** name a valid memory resource of the type specified by `handleType`.
- If `handleType` is not `0` and `name` is `NULL`, `handle` **must** be a valid handle of the type specified by `handleType`.
- If `handle` is not `NULL`, `name` must be `NULL`.
- If `handle` is not `NULL`, it **must** obey any requirements listed for `handleType` in [external memory handle types compatibility](#).
- If `name` is not `NULL`, it **must** obey any requirements listed for `handleType` in [external memory handle types compatibility](#).

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHR`
- If `handleType` is not `0`, `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

To export a Windows handle representing the underlying resources of a Vulkan device memory object, call:

```
VkResult vkGetMemoryWin32HandleKHR(  
    VkDevice                               device,  
    const VkMemoryGetWin32HandleInfoKHR* pGetWin32HandleInfo,  
    HANDLE*                                pHandle);
```

- `device` is the logical device that created the device memory being exported.
- `pGetWin32HandleInfo` is a pointer to a `VkMemoryGetWin32HandleInfoKHR` structure containing parameters of the export operation.

- `pHandle` will return the Windows handle representing the underlying resources of the device memory object.

For handle types defined as NT handles, the handles returned by `vkGetMemoryWin32HandleKHR` are owned by the application. To avoid leaking resources, the application **must** release ownership of them using the `CloseHandle` system call when they are no longer needed.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pGetWin32HandleInfo` **must** be a valid pointer to a valid `VkMemoryGetWin32HandleInfoKHR` structure
- `pHandle` **must** be a valid pointer to a `HANDLE` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkMemoryGetWin32HandleInfoKHR` structure is defined as:

```
typedef struct VkMemoryGetWin32HandleInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkDeviceMemory             memory;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkMemoryGetWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memory` is the memory object from which the handle will be exported.
- `handleType` is the type of handle requested.

The properties of the handle returned depend on the value of `handleType`. See `VkExternalMemoryHandleTypeFlagBits` for a description of the properties of the defined external memory handle types.

## Valid Usage

- `handleType` **must** have been included in `VkExportMemoryAllocateInfo::handleTypes` when `memory` was created.
- If `handleType` is defined as an NT handle, `vkGetMemoryWin32HandleKHR` **must** be called no more than once for each valid unique combination of `memory` and `handleType`.
- `handleType` **must** be defined as an NT handle or a global share handle.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_GET_WIN32_HANDLE_INFO_KHR`
- `pNext` **must** be `NULL`
- `memory` **must** be a valid `VkDeviceMemory` handle
- `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

Windows memory handles compatible with Vulkan **may** also be created by non-Vulkan APIs using methods beyond the scope of this specification. To determine the correct parameters to use when importing such handles, call:

```
VkResult vkGetMemoryWin32HandlePropertiesKHR(  
    VkDevice                                     device,  
    VkExternalMemoryHandleTypeFlagBits           handleType,  
    HANDLE                                       handle,  
    VkMemoryWin32HandlePropertiesKHR*            pMemoryWin32HandleProperties);
```

- `device` is the logical device that will be importing `handle`.
- `handleType` is the type of the handle `handle`.
- `handle` is the handle which will be imported.
- `pMemoryWin32HandleProperties` will return properties of `handle`.

## Valid Usage

- `handle` **must** be an external memory handle created outside of the Vulkan API.
- `handleType` **must** not be one of the handle types defined as opaque.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value
- `pMemoryWin32HandleProperties` **must** be a valid pointer to a `VkMemoryWin32HandlePropertiesKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkMemoryWin32HandlePropertiesKHR` structure returned is defined as:

```
typedef struct VkMemoryWin32HandlePropertiesKHR {
    VkStructureType sType;
    void*           pNext;
    uint32_t         memoryTypeBits;
} VkMemoryWin32HandlePropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memoryTypeBits` is a bitmask containing one bit set for every memory type which the specified windows handle **can** be imported as.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHR`
- `pNext` **must** be `NULL`

To import memory from a POSIX file descriptor handle, add a `VkImportMemoryFdInfoKHR` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure. The `VkImportMemoryFdInfoKHR` structure is defined as:

```

typedef struct VkImportMemoryFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    int fd;
} VkImportMemoryFdInfoKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **handleType** specifies the handle type of **fd**.
- **fd** is the external handle to import.

Importing memory from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application **must** not perform any operations on the file descriptor after a successful import.

Applications **can** import the same underlying memory into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance. In all cases, each import operation **must** create a distinct **VkDeviceMemory** object.

## Valid Usage

- If **handleType** is not **0**, it **must** be supported for import, as reported by **VkExternalImageFormatProperties** or **VkExternalBufferProperties**.
- The memory from which **fd** was exported **must** have been created on the same underlying physical device as **device**.
- If **handleType** is not **0**, it **must** be defined as a POSIX file descriptor handle.
- If **handleType** is not **0**, **fd** **must** be a valid handle of the type specified by **handleType**.
- The memory represented by **fd** **must** have been created from a physical device and driver that is compatible with **device** and **handleType**, as described in [External memory handle types compatibility](#).
- **fd** **must** obey any requirements listed for **handleType** in [external memory handle types compatibility](#).

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_IMPORT\_MEMORY\_FD\_INFO\_KHR**
- If **handleType** is not **0**, **handleType** **must** be a valid **VkExternalMemoryHandleTypeFlagBits** value

To export a POSIX file descriptor representing the underlying resources of a Vulkan device memory object, call:

```

VkResult vkGetMemoryFdKHR(
    VkDevice device,
    const VkMemoryGetFdInfoKHR* pGetFdInfo,
    int* pFd);

```

- `device` is the logical device that created the device memory being exported.
- `pGetFdInfo` is a pointer to a `VkMemoryGetFdInfoKHR` structure containing parameters of the export operation.
- `pFd` will return a file descriptor representing the underlying resources of the device memory object.

Each call to `vkGetMemoryFdKHR` **must** create a new file descriptor and transfer ownership of it to the application. To avoid leaking resources, the application **must** release ownership of the file descriptor using the `close` system call when it is no longer needed, or by importing a Vulkan memory object from it. Where supported by the operating system, the implementation **must** set the file descriptor to be closed automatically when an `execve` system call is made.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pGetFdInfo` **must** be a valid pointer to a valid `VkMemoryGetFdInfoKHR` structure
- `pFd` **must** be a valid pointer to an `int` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkMemoryGetFdInfoKHR` structure is defined as:

```

typedef struct VkMemoryGetFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkMemoryGetFdInfoKHR;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `memory` is the memory object from which the handle will be exported.
- `handleType` is the type of handle requested.

The properties of the file descriptor exported depend on the value of `handleType`. See `VkExternalMemoryHandleTypeFlagBits` for a description of the properties of the defined external memory handle types.

*Note*



The size of the exported file **may** be larger than the size requested by `VkMemoryAllocateInfo::allocationSize`. If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT`, then the application **can** query the file's actual size with `lseek(2)`.

## Valid Usage

- `handleType` **must** have been included in `VkExportMemoryAllocateInfo::handleTypes` when `memory` was created.
- `handleType` **must** be defined as a POSIX file descriptor handle.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR`
- `pNext` **must** be `NULL`
- `memory` **must** be a valid `VkDeviceMemory` handle
- `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

POSIX file descriptor memory handles compatible with Vulkan **may** also be created by non-Vulkan APIs using methods beyond the scope of this specification. To determine the correct parameters to use when importing such handles, call:

```
VkResult vkGetMemoryFdPropertiesKHR(
    VkDevice                                     device,
    VkExternalMemoryHandleTypeFlagBits           handleType,
    int                                           fd,
    VkMemoryFdPropertiesKHR*                    pMemoryFdProperties);
```

- `device` is the logical device that will be importing `fd`.
- `handleType` is the type of the handle `fd`.
- `fd` is the handle which will be imported.
- `pMemoryFdProperties` is a pointer to a `VkMemoryFdPropertiesKHR` structure in which the properties of the handle `fd` are returned.

## Valid Usage

- `fd` **must** be an external memory handle created outside of the Vulkan API.
- `handleType` **must** not be `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHR`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value
- `pMemoryFdProperties` **must** be a valid pointer to a `VkMemoryFdPropertiesKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkMemoryFdPropertiesKHR` structure returned is defined as:

```
typedef struct VkMemoryFdPropertiesKHR {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           memoryTypeBits;
} VkMemoryFdPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memoryTypeBits` is a bitmask containing one bit set for every memory type which the specified file descriptor **can** be imported as.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR`
- `pNext` **must** be `NULL`

To import memory from a host pointer, add a `VkImportMemoryHostPointerInfoEXT` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure. The `VkImportMemoryHostPointerInfoEXT` structure is defined as:

```

typedef struct VkImportMemoryHostPointerInfoEXT {
    VkStructureType           sType;
    const void*              pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    void*                  pHostPointer;
} VkImportMemoryHostPointerInfoEXT;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **handleType** specifies the handle type.
- **pHostPointer** is the host pointer to import from.

Importing memory from a host pointer shares ownership of the memory between the host and the Vulkan implementation. The application **can** continue to access the memory through the host pointer but it is the application's responsibility to synchronize device and non-device access to the underlying memory as defined in [Host Access to Device Memory Objects](#).

Applications **can** import the same underlying memory into multiple instances of Vulkan and multiple times into a given Vulkan instance. However, implementations **may** fail to import the same underlying memory multiple times into a given physical device due to platform constraints.

Importing memory from a particular host pointer **may** not be possible due to additional platform-specific restrictions beyond the scope of this specification in which case the implementation **must** fail the memory import operation with the error code **VK\_ERROR\_INVALID\_EXTERNAL\_HANDLE\_KHR**.

The application **must** ensure that the imported memory range remains valid and accessible for the lifetime of the imported memory object.

## Valid Usage

- If **handleType** is not **0**, it **must** be supported for import, as reported in [VkExternalMemoryProperties](#)
- If **handleType** is not **0**, it **must** be **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_HOST\_ALLOCATION\_BIT\_EXT** or **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_HOST\_MAPPED\_FOREIGN\_MEMORY\_BIT\_EXT**
- **pHostPointer** **must** be a pointer aligned to an integer multiple of [VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment](#)
- If **handleType** is **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_HOST\_ALLOCATION\_BIT\_EXT**, **pHostPointer** **must** be a pointer to **allocationSize** number of bytes of host memory, where **allocationSize** is the member of the [VkMemoryAllocateInfo](#) structure this structure is chained to
- If **handleType** is **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_HOST\_MAPPED\_FOREIGN\_MEMORY\_BIT\_EXT**, **pHostPointer** **must** be a pointer to **allocationSize** number of bytes of host mapped foreign memory, where **allocationSize** is the member of the [VkMemoryAllocateInfo](#) structure this structure is chained to

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT`
- `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

To determine the correct parameters to use when importing host pointers, call:

```
VkResult vkGetMemoryHostPointerPropertiesEXT(  
    VkDevice                                     device,  
    VkExternalMemoryHandleTypeFlagBits           handleType,  
    const void*                                    pHostPointer,  
    VkMemoryHostPointerPropertiesEXT*            pMemoryHostPointerProperties);
```

- `device` is the logical device that will be importing `pHostPointer`.
- `handleType` is the type of the handle `pHostPointer`.
- `pHostPointer` is the host pointer to import from.
- `pMemoryHostPointerProperties` is a pointer to a `VkMemoryHostPointerPropertiesEXT` structure in which the host pointer properties are returned.

## Valid Usage

- `handleType` **must** be `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`
- `pHostPointer` **must** be a pointer aligned to an integer multiple of `VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment`
- If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT`, `pHostPointer` **must** be a pointer to host memory
- If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`, `pHostPointer` **must** be a pointer to host mapped foreign memory

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value
- `pMemoryHostPointerProperties` **must** be a valid pointer to a `VkMemoryHostPointerPropertiesEXT` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_INVALID_EXTERNAL_HANDLE`

The `VkMemoryHostPointerPropertiesEXT` structure is defined as:

```
typedef struct VkMemoryHostPointerPropertiesEXT {
    VkStructureType    sType;
    void*             pNext;
    uint32_t          memoryTypeBits;
} VkMemoryHostPointerPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memoryTypeBits` is a bitmask containing one bit set for every memory type which the specified host pointer **can** be imported as.

The value returned by `memoryTypeBits` **must** only include bits that identify memory types which are host visible.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_HOST_POINTER_PROPERTIES_EXT`
- `pNext` **must** be `NULL`

To import memory created outside of the current Vulkan instance from an Android hardware buffer, add a `VkImportAndroidHardwareBufferInfoANDROID` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure. The `VkImportAndroidHardwareBufferInfoANDROID` structure is defined as:

```
typedef struct VkImportAndroidHardwareBufferInfoANDROID {
    VkStructureType          sType;
    const void*              pNext;
    struct AHardwareBuffer*   buffer;
} VkImportAndroidHardwareBufferInfoANDROID;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `buffer` is the Android hardware buffer to import.

If the `vkAllocateMemory` command succeeds, the implementation **must** acquire a reference to the imported hardware buffer, which it **must** release when the device memory object is freed. If the command fails, the implementation **must** not retain a reference.

## Valid Usage

- If `buffer` is not `NULL`, Android hardware buffers **must** be supported for import, as reported by `VkExternalImageFormatProperties` or `VkExternalBufferProperties`.
- If `buffer` is not `NULL`, it **must** be a valid Android hardware buffer object with `AHardwareBuffer_Desc::format` and `AHardwareBuffer_Desc::usage` compatible with Vulkan as described in [Android Hardware Buffers](#).

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_ANDROID_HARDWARE_BUFFER_INFO_ANDROID`
- `buffer` **must** be a valid pointer to an `AHardwareBuffer` value

To export an Android hardware buffer representing the underlying resources of a Vulkan device memory object, call:

```
VkResult vkGetMemoryAndroidHardwareBufferANDROID(  
    VkDevice                         device,  
    const VkMemoryGetAndroidHardwareBufferInfoANDROID* pInfo,  
    struct AHardwareBuffer**          pBuffer);
```

- `device` is the logical device that created the device memory being exported.
- `pInfo` is a pointer to a `VkMemoryGetAndroidHardwareBufferInfoANDROID` structure containing parameters of the export operation.
- `pBuffer` will return an Android hardware buffer representing the underlying resources of the device memory object.

Each call to `vkGetMemoryAndroidHardwareBufferANDROID` **must** return an Android hardware buffer with a new reference acquired in addition to the reference held by the `VkDeviceMemory`. To avoid leaking resources, the application **must** release the reference by calling `AHardwareBuffer_release` when it is no longer needed. When called with the same handle in `VkMemoryGetAndroidHardwareBufferInfoANDROID::memory`, `vkGetMemoryAndroidHardwareBufferANDROID` **must** return the same Android hardware buffer object. If the device memory was created by importing an Android hardware buffer, `vkGetMemoryAndroidHardwareBufferANDROID` **must** return that same Android hardware buffer object.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkMemoryGetAndroidHardwareBufferInfoANDROID` structure
- `pBuffer` **must** be a valid pointer to a valid pointer to an `AHardwareBuffer` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkMemoryGetAndroidHardwareBufferInfoANDROID` structure is defined as:

```
typedef struct VkMemoryGetAndroidHardwareBufferInfoANDROID {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceMemory     memory;
} VkMemoryGetAndroidHardwareBufferInfoANDROID;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memory` is the memory object from which the Android hardware buffer will be exported.

## Valid Usage

- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` **must** have been included in `VkExportMemoryAllocateInfo::handleTypes` when `memory` was created.
- If the `pNext` chain of the `VkMemoryAllocateInfo` used to allocate `memory` included a `VkMemoryDedicatedAllocateInfo` with non-`NULL` `image` member, then that `image` **must** already be bound to `memory`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_GET_ANDROID_HARDWARE_BUFFER_INFO_ANDROID`
- `pNext` **must** be `NULL`
- `memory` **must** be a valid `VkDeviceMemory` handle

To determine the memory parameters to use when importing an Android hardware buffer, call:

```
VkResult vkGetAndroidHardwareBufferPropertiesANDROID(  
    VkDevice device,  
    const struct AHardwareBuffer* buffer,  
    VkAndroidHardwareBufferPropertiesANDROID* pProperties);
```

- `device` is the logical device that will be importing `buffer`.
- `buffer` is the Android hardware buffer which will be imported.
- `pProperties` is a pointer to a `VkAndroidHardwareBufferPropertiesANDROID` structure in which the properties of `buffer` are returned.

## Valid Usage

- `buffer` **must** be a valid Android hardware buffer object with at least one of the `AHARDWAREBUFFER_USAGE_GPU_*` flags in its `AHardwareBuffer_Desc::usage`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `buffer` **must** be a valid pointer to a valid `AHardwareBuffer` value
- `pProperties` **must** be a valid pointer to a `VkAndroidHardwareBufferPropertiesANDROID` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR`

The `VkAndroidHardwareBufferPropertiesANDROID` structure returned is defined as:

```

typedef struct VkAndroidHardwareBufferPropertiesANDROID {
    VkStructureType      sType;
    void*              pNext;
    VkDeviceSize         allocationSize;
    uint32_t             memoryTypeBits;
} VkAndroidHardwareBufferPropertiesANDROID;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **allocationSize** is the size of the external memory
- **memoryTypeBits** is a bitmask containing one bit set for every memory type which the specified Android hardware buffer **can** be imported as.

### Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_PROPERTIES_ANDROID`
- **pNext** **must** be **NULL** or a pointer to a valid instance of `VkAndroidHardwareBufferFormatPropertiesANDROID`

To obtain format properties of an Android hardware buffer, include an instance of `VkAndroidHardwareBufferFormatPropertiesANDROID` in the **pNext** chain of the `VkAndroidHardwareBufferPropertiesANDROID` instance passed to `vkGetAndroidHardwareBufferPropertiesANDROID`. This structure is defined as:

```

typedef struct VkAndroidHardwareBufferFormatPropertiesANDROID {
    VkStructureType          sType;
    void*                  pNext;
    VkFormat                format;
    uint64_t                externalFormat;
    VkFormatFeatureFlags    formatFeatures;
    VkComponentMapping       samplerYcbcrConversionComponents;
    VkSamplerYcbcrModelConversion suggestedYcbcrModel;
    VkSamplerYcbcrRange     suggestedYcbcrRange;
    VkChromaLocation        suggestedXChromaOffset;
    VkChromaLocation        suggestedYChromaOffset;
} VkAndroidHardwareBufferFormatPropertiesANDROID;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **format** is the Vulkan format corresponding to the Android hardware buffer's format, or `VK_FORMAT_UNDEFINED` if there is not an equivalent Vulkan format.
- **externalFormat** is an implementation-defined external format identifier for use with `VkExternalFormatANDROID`. It **must** not be zero.

- `formatFeatures` describes the capabilities of this external format when used with an image bound to memory imported from `buffer`.
- `samplerYcbcrConversionComponents` is the component swizzle that **should** be used in `VkSamplerYcbcrConversionCreateInfo`.
- `suggestedYcbcrModel` is a suggested color model to use in the `VkSamplerYcbcrConversionCreateInfo`.
- `suggestedYcbcrRange` is a suggested numerical value range to use in `VkSamplerYcbcrConversionCreateInfo`.
- `suggestedXChromaOffset` is a suggested X chroma offset to use in `VkSamplerYcbcrConversionCreateInfo`.
- `suggestedYChromaOffset` is a suggested Y chroma offset to use in `VkSamplerYcbcrConversionCreateInfo`.

If the Android hardware buffer has one of the formats listed in the [Format Equivalence table](#), then `format` **must** have the equivalent Vulkan format listed in the table. Otherwise, `format` **may** be `VK_FORMAT_UNDEFINED`, indicating the Android hardware buffer **can** only be used with an external format.

The `formatFeatures` member **must** include `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` and at least one of `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT` or `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`, and **should** include `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT`.

*Note*

The `formatFeatures` member only indicates the features available when using an [external-format image](#) created from the Android hardware buffer. Images from Android hardware buffers with a format other than `VK_FORMAT_UNDEFINED` are subject to the format capabilities obtained from `vkGetPhysicalDeviceFormatProperties2` and `vkGetPhysicalDeviceImageFormatProperties2` with appropriate parameters. These sets of features are independent of each other, e.g. the external format will support sampler Y'CbCr conversion even if the non-external format does not, and writing to non-external format images is possible but writing to external format images is not.



Android hardware buffers with the same external format **must** have the same support for `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`, `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT`, `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`, `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT`, `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT`, and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT` in `formatFeatures`. Other format features **may** differ between Android hardware buffers that have the same external format. This allows applications to use the same `VkSamplerYcbcrConversion` object (and samplers and pipelines created from them) for any Android hardware buffers that have the same external format.

If `format` is not `VK_FORMAT_UNDEFINED`, then the value of `samplerYcbcrConversionComponents` **must** be valid when used as the `components` member of `VkSamplerYcbcrConversionCreateInfo` with that format. If `format` is `VK_FORMAT_UNDEFINED`, all members of `samplerYcbcrConversionComponents` **must** be `VK_COMPONENT_SWIZZLE_IDENTITY`.

Implementations **may** not always be able to determine the color model, numerical range, or chroma offsets of the image contents, so the values in `VkAndroidHardwareBufferFormatPropertiesANDROID` are only suggestions. Applications **should** treat these values as sensible defaults to use in the absence of more reliable information obtained through some other means. If the underlying physical device is also usable via OpenGL ES with the `GL_OES_EGL_image_external` extension, the implementation **should** suggest values that will produce similar sampled values as would be obtained by sampling the same external image via `samplerExternalOES` in OpenGL ES using equivalent sampler parameters.

*Note*



Since `GL_OES_EGL_image_external` does not require the same sampling and conversion calculations as Vulkan does, achieving identical results between APIs **may** not be possible on some implementations.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_FORMAT_PROPERTIES_ANDROID`

When allocating memory that **may** be exported to another process or Vulkan instance, add a `VkExportMemoryAllocateInfoNV` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure, specifying the handle types that **may** be exported.

The `VkExportMemoryAllocateInfoNV` structure is defined as:

```
typedef struct VkExportMemoryAllocateInfoNV {
    VkStructureType             sType;
    const void*                 pNext;
    VkExternalMemoryHandleTypeFlagsNV handleTypes;
} VkExportMemoryAllocateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` specifying one or more memory handle types that **may** be exported. Multiple handle types **may** be requested for the same allocation as long as they are compatible, as reported by `vkGetPhysicalDeviceExternalImageFormatPropertiesNV`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV`
- `handleTypes` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values

When `VkExportMemoryAllocateInfoNV::handleTypes` includes `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV`, add a `VkExportMemoryWin32HandleInfoNV` to the `pNext` chain of the `VkExportMemoryAllocateInfoNV` structure to specify security attributes and access rights for the memory object's external handle.

The `VkExportMemoryWin32HandleInfoNV` structure is defined as:

```
typedef struct VkExportMemoryWin32HandleInfoNV {
    VkStructureType           sType;
    const void*               pNext;
    const SECURITY_ATTRIBUTES* pAttributes;
    DWORD                     dwAccess;
} VkExportMemoryWin32HandleInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pAttributes` is a pointer to a Windows `SECURITY_ATTRIBUTES` structure specifying security attributes of the handle.
- `dwAccess` is a `DWORD` specifying access rights of the handle.

If this structure is not present, or if `pAttributes` is set to `NULL`, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in the MSDN documentation for “Synchronization Object Security and Access Rights”<sup>1</sup>. Further, if the structure is not present, the access rights will be

`DXGI_SHARED_RESOURCE_READ | DXGI_SHARED_RESOURCE_WRITE`

1

<https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-object-security-and-access-rights>

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_NV`
- If `pAttributes` is not `NULL`, `pAttributes` **must** be a valid pointer to a valid `SECURITY_ATTRIBUTES` value

To import memory created on the same physical device but outside of the current Vulkan instance,

add a `VkImportMemoryWin32HandleInfoNV` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure, specifying a handle to and the type of the memory.

The `VkImportMemoryWin32HandleInfoNV` structure is defined as:

```
typedef struct VkImportMemoryWin32HandleInfoNV {
    VkStructureType           sType;
    const void*               pNext;
    VkExternalMemoryHandleTypeFlagsNV handleType;
    HANDLE                   handle;
} VkImportMemoryWin32HandleInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleType` is `0` or a `VkExternalMemoryHandleTypeFlagBitsNV` value specifying the type of memory handle in `handle`.
- `handle` is a Windows `HANDLE` referring to the memory.

If `handleType` is `0`, this structure is ignored by consumers of the `VkMemoryAllocateInfo` structure it is chained from.

## Valid Usage

- `handleType` **must** not have more than one bit set.
- `handle` **must** be a valid handle to memory, obtained as specified by `handleType`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_NV`
- `handleType` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values

Bits which **can** be set in `handleType` are:

Possible values of `VkImportMemoryWin32HandleInfoNV::handleType`, specifying the type of an external memory handle, are:

```

typedef enum VkExternalMemoryHandleTypeFlagBitsNV {
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV = 0x00000001,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_NV = 0x00000002,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_BIT_NV = 0x00000004,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_KMT_BIT_NV = 0x00000008,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_FLAG_BITS_MAX_ENUM_NV = 0x7FFFFFFF
} VkExternalMemoryHandleTypeFlagBitsNV;

```

- **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_WIN32\_KMT\_BIT\_NV** specifies a handle to memory returned by [vkGetMemoryWin32HandleNV](#).
- **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_WIN32\_BIT\_NV** specifies a handle to memory returned by [vkGetMemoryWin32HandleNV](#), or one duplicated from such a handle using [DuplicateHandle\(\)](#).
- **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_D3D11\_IMAGE\_BIT\_NV** specifies a valid NT handle to memory returned by [IDXGIResource1::CreateSharedHandle](#), or a handle duplicated from such a handle using [DuplicateHandle\(\)](#).
- **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_D3D11\_IMAGE\_KMT\_BIT\_NV** specifies a handle to memory returned by [IDXGIResource::GetSharedHandle\(\)](#).

```
typedef VkFlags VkExternalMemoryHandleTypeFlagsNV;
```

**VkExternalMemoryHandleTypeFlagsNV** is a bitmask type for setting a mask of zero or more **VkExternalMemoryHandleTypeFlagBitsNV**.

To retrieve the handle corresponding to a device memory object created with [VkExportMemoryAllocateInfoNV::handleTypes](#) set to include **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_WIN32\_BIT\_NV** or **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_WIN32\_KMT\_BIT\_NV**, call:

```

VkResult vkGetMemoryWin32HandleNV(
    VkDevice device,
    VkDeviceMemory memory,
    VkExternalMemoryHandleTypeFlagsNV handleType,
    HANDLE* pHandle);

```

- **device** is the logical device that owns the memory.
- **memory** is the **VkDeviceMemory** object.
- **handleType** is a bitmask of **VkExternalMemoryHandleTypeFlagBitsNV** containing a single bit specifying the type of handle requested.
- **handle** is a pointer to a Windows **HANDLE** in which the handle is returned.

## Valid Usage

- `handleType` **must** be a flag specified in `VkExportMemoryAllocateInfoNV::handleTypes` when allocating `memory`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- `handleType` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values
- `handleType` **must** not be `0`
- `pHandle` **must** be a valid pointer to a `HANDLE` value
- `memory` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

If the `pNext` chain of `VkMemoryAllocateInfo` includes a `VkMemoryAllocateFlagsInfo` structure, then that structure includes flags and a device mask controlling how many instances of the memory will be allocated.

The `VkMemoryAllocateFlagsInfo` structure is defined as:

```
typedef struct VkMemoryAllocateFlagsInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkMemoryAllocateFlags    flags;
    uint32_t                 deviceMask;
} VkMemoryAllocateFlagsInfo;
```

or the equivalent

```
typedef VkMemoryAllocateFlagsInfo VkMemoryAllocateFlagsInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkMemoryAllocateFlagBits` controlling the allocation.
- `deviceMask` is a mask of physical devices in the logical device, indicating that memory **must** be allocated on each device in the mask, if `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set in `flags`.

If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is not set, the number of instances allocated depends on whether `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set in the memory heap. If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set, then memory is allocated for every physical device in the logical device (as if `deviceMask` has bits set for all device indices). If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is not set, then a single instance of memory is allocated (as if `deviceMask` is set to one).

On some implementations, allocations from a multi-instance heap **may** consume memory on all physical devices even if the `deviceMask` excludes some devices. If `VkPhysicalDeviceGroupProperties::subsetAllocation` is `VK_TRUE`, then memory is only consumed for the devices in the device mask.

*Note*



In practice, most allocations on a multi-instance heap will be allocated across all physical devices. Unicast allocation support is an optional optimization for a minority of allocations.

## Valid Usage

- If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set, `deviceMask` **must** be a valid device mask.
- If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set, `deviceMask` **must** not be zero

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO`
- `flags` **must** be a valid combination of `VkMemoryAllocateFlagBits` values

Bits which **can** be set in `VkMemoryAllocateFlagsInfo::flags`, controlling device memory allocation, are:

```
typedef enum VkMemoryAllocateFlagBits {
    VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT = 0x00000001,
    VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR = 0x00000002,
    VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR = 0x00000004,
    VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT_KHR = VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT,
    VK_MEMORY_ALLOCATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkMemoryAllocateFlagBits;
```

or the equivalent

```
typedef VkMemoryAllocateFlagBits VkMemoryAllocateFlagBitsKHR;
```

- `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` specifies that memory will be allocated for the devices in `VkMemoryAllocateFlagsInfo::deviceMask`.
- `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR` specifies that the memory **can** be attached to a buffer object created with the `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR` bit set in `usage`, and that the memory handle **can** be used to retrieve an opaque address via `vkGetDeviceMemoryOpaqueCaptureAddressKHR`.
- `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR` specifies that the memory's address **can** be saved and reused on a subsequent run (e.g. for trace capture and replay), see `VkBufferOpaqueCaptureAddressCreateInfoKHR` for more detail.

```
typedef VkFlags VkMemoryAllocateFlags;
```

or the equivalent

```
typedef VkMemoryAllocateFlags VkMemoryAllocateFlagsKHR;
```

`VkMemoryAllocateFlags` is a bitmask type for setting a mask of zero or more `VkMemoryAllocateFlagBits`.

To request a specific device address for a memory allocation, add a `VkMemoryOpaqueCaptureAddressAllocateInfoKHR` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure. The `VkMemoryOpaqueCaptureAddressAllocateInfoKHR` structure is defined as:

```
typedef struct VkMemoryOpaqueCaptureAddressAllocateInfoKHR {
    VkStructureType sType;
    const void*     pNext;
    uint64_t        opaqueCaptureAddress;
} VkMemoryOpaqueCaptureAddressAllocateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `opaqueCaptureAddress` is the opaque capture address requested for the memory allocation.

If `opaqueCaptureAddress` is zero, no specific address is requested.

If `opaqueCaptureAddress` is not zero, it **should** be an address retrieved from `vkGetDeviceMemoryOpaqueCaptureAddressKHR` on an identically created memory allocation on the same implementation.

*Note*

In most cases, it is expected that a non-zero `opaqueAddress` is an address retrieved from `vkGetDeviceMemoryOpaqueCaptureAddressKHR` on an identically created memory allocation. If this is not the case, it likely that `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR` errors will occur.

This is, however, not a strict requirement because trace capture/replay tools may need to adjust memory allocation parameters for imported memory.

If this structure is not present, it is as if `opaqueCaptureAddress` is zero.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO_KHR`

To free a memory object, call:

```
void vkFreeMemory(  
    VkDevice device,  
    VkDeviceMemory memory,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that owns the memory.
- `memory` is the `VkDeviceMemory` object to be freed.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

Before freeing a memory object, an application **must** ensure the memory object is no longer in use by the device—for example by command buffers in the *pending state*. Memory **can** be freed whilst still bound to resources, but those resources **must** not be used afterwards. If there are still any bound images or buffers, the memory **may** not be immediately released by the implementation, but **must** be released by the time all bound images and buffers have been destroyed. Once memory is released, it is returned to the heap from which it was allocated.

How memory objects are bound to Images and Buffers is described in detail in the [Resource Memory Association](#) section.

If a memory object is mapped at the time it is freed, it is implicitly unmapped.

*Note*

As described [below](#), host writes are not implicitly flushed when the memory object is unmapped, but the implementation **must** guarantee that writes that have not been flushed do not affect any other memory.

## Valid Usage

- All submitted commands that refer to `memory` (via images or buffers) **must** have completed execution

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `memory` is not `VK_NULL_HANDLE`, `memory` **must** be a valid `VkDeviceMemory` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `memory` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `memory` **must** be externally synchronized

### 10.2.1. Host Access to Device Memory Objects

Memory objects created with `vkAllocateMemory` are not directly host accessible.

Memory objects created with the memory property `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` are considered *mappable*. Memory objects **must** be mappable in order to be successfully mapped on the host.

To retrieve a host virtual address pointer to a region of a mappable memory object, call:

```
VkResult vkMapMemory(  
    VkDevice                                device,  
    VkDeviceMemory                            memory,  
    VkDeviceSize                             offset,  
    VkDeviceSize                             size,  
    VkMemoryMapFlags                         flags,  
    void**                                   ppData);
```

- `device` is the logical device that owns the memory.
- `memory` is the `VkDeviceMemory` object to be mapped.
- `offset` is a zero-based byte offset from the beginning of the memory object.
- `size` is the size of the memory range to map, or `VK_WHOLE_SIZE` to map from `offset` to the end of the allocation.
- `flags` is reserved for future use.

- `ppData` is a pointer to a `void *` variable in which is returned a host-accessible pointer to the beginning of the mapped range. This pointer minus `offset` **must** be aligned to at least `VkPhysicalDeviceLimits::minMemoryMapAlignment`.

After a successful call to `vkMapMemory` the memory object `memory` is considered to be currently *host mapped*. It is an application error to call `vkMapMemory` on a memory object that is already host mapped.

*Note*



`vkMapMemory` will fail if the implementation is unable to allocate an appropriately sized contiguous virtual address range, e.g. due to virtual address space fragmentation or platform limits. In such cases, `vkMapMemory` **must** return `VK_ERROR_MEMORY_MAP_FAILED`. The application **can** improve the likelihood of success by reducing the size of the mapped range and/or removing unneeded mappings using `vkUnmapMemory`.

`vkMapMemory` does not check whether the device memory is currently in use before returning the host-accessible pointer. The application **must** guarantee that any previously submitted command that writes to this range has completed before the host reads from or writes to that range, and that any previously submitted command that reads from that range has completed before the host writes to that region (see [here](#) for details on fulfilling such a guarantee). If the device memory was allocated without the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` set, these guarantees **must** be made for an extended range: the application **must** round down the start of the range to the nearest multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize`, and round the end of the range up to the nearest multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize`.

While a range of device memory is host mapped, the application is responsible for synchronizing both device and host access to that memory range.

*Note*



It is important for the application developer to become meticulously familiar with all of the mechanisms described in the chapter on [Synchronization and Cache Control](#) as they are crucial to maintaining memory access ordering.

## Valid Usage

- `memory` **must** not be currently host mapped
- `offset` **must** be less than the size of `memory`
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** be greater than `0`
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** be less than or equal to the size of the `memory` minus `offset`
- `memory` **must** have been created with a memory type that reports `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT`
- `memory` **must** not have been allocated with multiple instances.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- `flags` **must** be `0`
- `ppData` **must** be a valid pointer to a pointer value
- `memory` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `memory` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_MEMORY_MAP_FAILED`

```
typedef VkFlags VkMemoryMapFlags;
```

`VkMemoryMapFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

Two commands are provided to enable applications to work with non-coherent memory allocations: `vkFlushMappedMemoryRanges` and `vkInvalidateMappedMemoryRanges`.

### Note

If the memory object was created with the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` set, `vkFlushMappedMemoryRanges` and `vkInvalidateMappedMemoryRanges` are unnecessary and **may** have a performance cost. However, [availability and visibility operations](#) still need to be managed on the device. See the description of [host access types](#) for more information.



*Note*

While memory objects imported from a handle type of `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT` are inherently mapped to host address space, they are not considered to be host mapped device memory unless they are explicitly host mapped using `vkMapMemory`. That means flushing or invalidating host caches with respect to host accesses performed on such memory through the original host pointer specified at import time is the responsibility of the application and **must** be performed with appropriate synchronization primitives provided by the platform which are outside the scope of Vulkan. `vkFlushMappedMemoryRanges` and `vkInvalidateMappedMemoryRanges`, however, **can** still be used on such memory objects to synchronize host accesses performed through the host pointer of the host mapped device memory range returned by `vkMapMemory`.



To flush ranges of non-coherent memory from the host caches, call:

```
VkResult vkFlushMappedMemoryRanges(  
    VkDevice device,  
    uint32_t memoryRangeCount,  
    const VkMappedMemoryRange* pMemoryRanges);
```

- `device` is the logical device that owns the memory ranges.
- `memoryRangeCount` is the length of the `pMemoryRanges` array.
- `pMemoryRanges` is a pointer to an array of `VkMappedMemoryRange` structures describing the memory ranges to flush.

`vkFlushMappedMemoryRanges` guarantees that host writes to the memory ranges described by `pMemoryRanges` are made available to the host memory domain, such that they **can** be made available to the device memory domain via `memory domain operations` using the `VK_ACCESS_HOST_WRITE_BIT` access type.

Within each range described by `pMemoryRanges`, each set of `nonCoherentAtomSize` bytes in that range is flushed if any byte in that set has been written by the host since it was first host mapped, or the last time it was flushed. If `pMemoryRanges` includes sets of `nonCoherentAtomSize` bytes where no bytes have been written by the host, those bytes **must** not be flushed.

Unmapping non-coherent memory does not implicitly flush the host mapped memory, and host writes that have not been flushed **may** not ever be visible to the device. However, implementations **must** ensure that writes that have not been flushed do not become visible to any other memory.

*Note*



The above guarantee avoids a potential memory corruption in scenarios where host writes to a mapped memory object have not been flushed before the memory is unmapped (or freed), and the virtual address range is subsequently reused for a different mapping (or memory allocation).

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pMemoryRanges` **must** be a valid pointer to an array of `memoryRangeCount` valid `VkMappedMemoryRange` structures
- `memoryRangeCount` **must** be greater than `0`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

To invalidate ranges of non-coherent memory from the host caches, call:

```
VkResult vkInvalidateMappedMemoryRanges(  
    VkDevice device,  
    uint32_t memoryRangeCount,  
    const VkMappedMemoryRange* pMemoryRanges);
```

- `device` is the logical device that owns the memory ranges.
- `memoryRangeCount` is the length of the `pMemoryRanges` array.
- `pMemoryRanges` is a pointer to an array of `VkMappedMemoryRange` structures describing the memory ranges to invalidate.

`vkInvalidateMappedMemoryRanges` guarantees that device writes to the memory ranges described by `pMemoryRanges`, which have been made available to the host memory domain using the `VK_ACCESS_HOST_WRITE_BIT` and `VK_ACCESS_HOST_READ_BIT` access types, are made visible to the host. If a range of non-coherent memory is written by the host and then invalidated without first being flushed, its contents are undefined.

Within each range described by `pMemoryRanges`, each set of `nonCoherentAtomSize` bytes in that range is invalidated if any byte in that set has been written by the device since it was first host mapped, or the last time it was invalidated.



### Note

Mapping non-coherent memory does not implicitly invalidate that memory.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pMemoryRanges` **must** be a valid pointer to an array of `memoryRangeCount` valid `VkMappedMemoryRange` structures
- `memoryRangeCount` **must** be greater than `0`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkMappedMemoryRange` structure is defined as:

```
typedef struct VkMappedMemoryRange {  
    VkStructureType    sType;  
    const void*        pNext;  
    VkDeviceMemory     memory;  
    VkDeviceSize       offset;  
    VkDeviceSize       size;  
} VkMappedMemoryRange;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memory` is the memory object to which this range belongs.
- `offset` is the zero-based byte offset from the beginning of the memory object.
- `size` is either the size of range, or `VK_WHOLE_SIZE` to affect the range from `offset` to the end of the current mapping of the allocation.

## Valid Usage

- `memory` **must** be currently host mapped
- If `size` is not equal to `VK_WHOLE_SIZE`, `offset` and `size` **must** specify a range contained within the currently mapped range of `memory`
- If `size` is equal to `VK_WHOLE_SIZE`, `offset` **must** be within the currently mapped range of `memory`
- If `size` is equal to `VK_WHOLE_SIZE`, the end of the current mapping of `memory` **must** be a multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize` bytes from the beginning of the memory object.
- `offset` **must** be a multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize`
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** either be a multiple of `VkPhysicalDeviceLimits::nonCoherentAtomSize`, or `offset` plus `size` **must** equal the size of `memory`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE`
- `pNext` **must** be `NULL`
- `memory` **must** be a valid `VkDeviceMemory` handle

To unmap a memory object once host access to it is no longer needed by the application, call:

```
void vkUnmapMemory(  
    VkDevice           device,  
    VkDeviceMemory     memory);
```

- `device` is the logical device that owns the memory.
- `memory` is the memory object to be unmapped.

## Valid Usage

- `memory` **must** be currently host mapped

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- `memory` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `memory` **must** be externally synchronized

### 10.2.2. Lazily Allocated Memory

If the memory object is allocated from a heap with the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit set, that object's backing memory **may** be provided by the implementation lazily. The actual committed size of the memory **may** initially be as small as zero (or as large as the requested size), and monotonically increases as additional memory is needed.

A memory type with this flag set is only allowed to be bound to a `VkImage` whose usage flags include `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`.

*Note*



Using lazily allocated memory objects for framebuffer attachments that are not needed once a render pass instance has completed **may** allow some implementations to never allocate memory for such attachments.

To determine the amount of lazily-allocated memory that is currently committed for a memory object, call:

```
void vkGetDeviceMemoryCommitment(  
    VkDevice                                     device,  
    VkDeviceMemory                                memory,  
    VkDeviceSize*                                 pCommittedMemoryInBytes);
```

- `device` is the logical device that owns the memory.
- `memory` is the memory object being queried.
- `pCommittedMemoryInBytes` is a pointer to a `VkDeviceSize` value in which the number of bytes currently committed is returned, on success.

The implementation **may** update the commitment at any time, and the value returned by this query **may** be out of date.

The implementation guarantees to allocate any committed memory from the `heapIndex` indicated by the memory type that the memory object was created with.

### Valid Usage

- `memory` **must** have been created with a memory type that reports `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- `pCommittedMemoryInBytes` **must** be a valid pointer to a `VkDeviceSize` value
- `memory` **must** have been created, allocated, or retrieved from `device`

### 10.2.3. Protected Memory

*Protected memory* divides device memory into protected device memory and unprotected device memory.

Protected memory adds the following concepts:

- Memory:
  - Unprotected device memory, which **can** be visible to the device and **can** be visible to the host
  - Protected device memory, which **can** be visible to the device but **must** not be visible to the host
- Resources:
  - Unprotected images and unprotected buffers, to which unprotected memory **can** be bound
  - Protected images and protected buffers, to which protected memory **can** be bound
- Command buffers:
  - Unprotected command buffers, which **can** be submitted to a device queue to execute unprotected queue operations
  - Protected command buffers, which **can** be submitted to a protected-capable device queue to execute protected queue operations
- Device queues:
  - Unprotected device queues, to which unprotected command buffers **can** be submitted
  - Protected-capable device queues, to which unprotected command buffers or protected command buffers **can** be submitted
- Queue submissions
  - Unprotected queue submissions, through which unprotected command buffers **can** be submitted
  - Protected queue submissions, through which protected command buffers **can** be submitted
- Queue operations
  - Unprotected queue operations
  - Protected queue operations

## Protected Memory Access Rules

If `VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault` is `VK_FALSE`, applications **must** not perform any of the following operations:

- Write to unprotected memory within protected queue operations.
- Access protected memory within protected queue operations other than in framebuffer-space pipeline stages, the compute shader stage, or the transfer stage.
- Perform a query within protected queue operations.
- Execute an indirect command within protected queue operations.

If `VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault` is `VK_TRUE`, these operations are valid, but reads will return undefined values, and writes will either be dropped or store undefined values.

Whether these operations are valid or not, or if any other invalid usage is performed, the implementation **must** guarantee that:

- Protected device memory **must** never be visible to the host.
- Values written to unprotected device memory **must** not be a function of values from protected memory.

### 10.2.4. External Memory Handle Types

#### Android Hardware Buffer

Android's NDK defines `AHardwareBuffer` objects, which represent device memory that is shareable across processes and that **can** be accessed by a variety of media APIs and the hardware used to implement them. These Android hardware buffer objects **may** be imported into `VkDeviceMemory` objects for access via Vulkan, or exported from Vulkan.

To remove an unnecessary compile-time dependency, an incomplete type definition of `AHardwareBuffer` is provided in the Vulkan headers:

```
struct AHardwareBuffer;
```

The actual `AHardwareBuffer` type is defined in Android NDK headers.

#### Note

The NDK format, usage, and size/dimensions of an `AHardwareBuffer` object can be obtained with the `AHardwareBuffer_describe` function. While Android hardware buffers can be imported to or exported from Vulkan without using that function, valid usage and implementation behavior is defined in terms of the `AHardwareBuffer_Desc` properties it returns.



Android hardware buffer objects are reference-counted using Android NDK functions outside of the scope of this specification. A `VkDeviceMemory` imported from an Android hardware buffer or

that **can** be exported to an Android hardware buffer **must** acquire a reference to its [AHardwareBuffer](#) object, and **must** release this reference when the device memory is freed. During the host execution of a Vulkan command that has an Android hardware buffer as a parameter (including indirect parameters via [pNext](#) chains), the application **must** not decrement the Android hardware buffer's reference count to zero.

Android hardware buffers **can** be mapped and unmapped for CPU access using the NDK functions. These lock and unlock APIs are considered to acquire and release ownership of the Android hardware buffer, and applications **must** follow the rules described in [External Resource Sharing](#) to transfer ownership between the Vulkan instance and these native APIs.

Android hardware buffers **can** be shared with external APIs and Vulkan instances on the same device, and also with foreign devices. When transferring ownership of the Android hardware buffer, the external and foreign special queue families described in [Queue Family Ownership Transfer](#) are not identical. All APIs which produce or consume Android hardware buffers are considered to use foreign devices, except OpenGL ES contexts and Vulkan logical devices that have matching device and driver UUIDs. Implementations **may** treat a transfer to or from the foreign queue family as if it were a transfer to or from the external queue family when the Android hardware buffer's usage only permits it to be used on the same physical device.

### Android Hardware Buffer Optimal Usages

Vulkan buffer and image usage flags do not correspond exactly to Android hardware buffer usage flags. When allocating Android hardware buffers with non-Vulkan APIs, if any [AHARDWAREBUFFER\\_USAGE\\_GPU\\_\\*](#) usage bits are included, by default the allocator **must** allocate the memory in such a way that it supports Vulkan usages and creation flags in the [usage equivalence table](#) which do not have Android hardware buffer equivalents.

The [VkAndroidHardwareBufferUsageANDROID](#) structure **can** be attached to the [pNext](#) chain of a [VkImageFormatProperties2](#) instance passed to [vkGetPhysicalDeviceImageFormatProperties2](#) to obtain optimal Android hardware buffer usage flags for specific Vulkan resource creation parameters. Some usage flags returned by these commands are **required** based on the input parameters, but additional vendor-specific usage flags ([AHARDWAREBUFFER\\_USAGE\\_VENDOR\\_\\*](#)) **may** also be returned. Any Android hardware buffer allocated with these vendor-specific usage flags and imported to Vulkan **must** only be bound to resources created with parameters that are a subset of the parameters used to obtain the Android hardware buffer usage, since the memory **may** have been allocated in a way incompatible with other parameters. If an Android hardware buffer is successfully allocated with additional non-vendor-specific usage flags in addition to the recommended usage, it **must** support being used in the same ways as an Android hardware buffer allocated with only the recommended usage, and also in ways indicated by the additional usage.

### Android Hardware Buffer External Formats

Android hardware buffers **may** represent images using implementation-specific formats, layouts, color models, etc., which do not have Vulkan equivalents. Such *external formats* are commonly used by external image sources such as video decoders or cameras. Vulkan **can** import Android hardware buffers that have external formats, but since the image contents are in an undiscoverable and possibly proprietary representation, images with external formats **must** only be used as sampled images, **must** only be sampled with a sampler that has Y'C<sub>B</sub>C<sub>R</sub> conversion

enabled, and **must** have optimal tiling.

Images that will be backed by an Android hardware buffer **can** use an external format by setting `VkImageCreateInfo::format` to `VK_FORMAT_UNDEFINED` and including an instance of `VkExternalFormatANDROID` in the `pNext` chain. Images **can** be created with an external format even if the Android hardware buffer has a format which has an equivalent Vulkan format to enable consistent handling of images from sources that might use either category of format. However, all images created with an external format are subject to the valid usage requirements associated with external formats, even if the Android hardware buffer's format has a Vulkan equivalent. The external format of an Android hardware buffer **can** be obtained by passing an instance of `VkAndroidHardwareBufferFormatPropertiesANDROID` to `vkGetAndroidHardwareBufferPropertiesANDROID`.

### Android Hardware Buffer Image Resources

Android hardware buffers have intrinsic width, height, format, and usage properties, so Vulkan images bound to memory imported from an Android hardware buffer **must** use dedicated allocations: `VkMemoryDedicatedRequirements::requiresDedicatedAllocation` **must** be `VK_TRUE` for images created with `VkExternalMemoryImageCreateInfo::handleTypes` that includes `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`. When creating an image that will be bound to an imported Android hardware buffer, the image creation parameters **must** be equivalent to the `AHardwareBuffer` properties as described by the valid usage of `VkMemoryAllocateInfo`. Similarly, device memory allocated for a dedicated image **must** not be exported to an Android hardware buffer until it has been bound to that image, and the implementation **must** return an Android hardware buffer with properties derived from the image:

- The `width` and `height` members of `AHardwareBuffer_Desc` **must** be the same as the `width` and `height` members of `VkImageCreateInfo::extent`, respectively.
- The `layers` member of `AHardwareBuffer_Desc` **must** be the same as the `arrayLayers` member of `VkImageCreateInfo`.
- The `format` member of `AHardwareBuffer_Desc` **must** be equivalent to `VkImageCreateInfo::format` as defined by `AHardwareBuffer Format Equivalence`.
- The `usage` member of `AHardwareBuffer_Desc` **must** include bits corresponding to bits included in `VkImageCreateInfo::usage` and `VkImageCreateInfo::flags` where such a correspondence exists according to `AHardwareBuffer Usage Equivalence`. It **may** also include additional usage bits, including vendor-specific usages. Presence of vendor usage bits **may** make the Android hardware buffer only usable in ways indicated by the image creation parameters, even when used outside Vulkan, in a similar way that allocating the Android hardware buffer with usage returned in `VkAndroidHardwareBufferUsageANDROID` does.

Implementations **may** support fewer combinations of image creation parameters for images with Android hardware buffer external handle type than for non-external images. Support for a given set of parameters **can** be determined by passing `VkExternalImageFormatProperties` to `vkGetPhysicalDeviceImageFormatProperties2` with `handleType` set to `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`. Any Android hardware buffer successfully allocated outside Vulkan with usage that includes `AHARDWAREBUFFER_USAGE_GPU_*` **must** be supported when using equivalent Vulkan image parameters. If a given choice of image parameters are supported for import, they **can** also be used to create an image and memory that

will be exported to an Android hardware buffer.

*Table 13. AHardwareBuffer Format Equivalence*

AHardwareBuffer Format	Vulkan Format
AHARDWAREBUFFER_FORMAT_R8G8B8A8_UNORM	VK_FORMAT_R8G8B8A8_UNORM
AHARDWAREBUFFER_FORMAT_R8G8B8X8_UNORM <sup>1</sup>	VK_FORMAT_R8G8B8A8_UNORM
AHARDWAREBUFFER_FORMAT_R8G8B8_UNORM	VK_FORMAT_R8G8B8_UNORM
AHARDWAREBUFFER_FORMAT_R5G6B5_UNORM	VK_FORMAT_R5G6B5_UNORM_PACK16
AHARDWAREBUFFER_FORMAT_R16G16B16A16_FLOAT	VK_FORMAT_R16G16B16A16_SFLOAT
AHARDWAREBUFFER_FORMAT_R10G10B10A2_UNORM	VK_FORMAT_A2B10G10R10_UNORM_PACK32
AHARDWAREBUFFER_FORMAT_D16_UNORM	VK_FORMAT_D16_UNORM
AHARDWAREBUFFER_FORMAT_D24_UNORM	VK_FORMAT_X8_D24_UNORM_PACK32
AHARDWAREBUFFER_FORMAT_D24_UNORM_S8_UINT	VK_FORMAT_D24_UNORM_S8_UINT
AHARDWAREBUFFER_FORMAT_D32_FLOAT	VK_FORMAT_D32_SFLOAT
AHARDWAREBUFFER_FORMAT_D32_FLOAT_S8_UINT	VK_FORMAT_D32_SFLOAT_S8_UINT
AHARDWAREBUFFER_FORMAT_S8_UINT	VK_FORMAT_S8_UINT

*Table 14. AHardwareBuffer Usage Equivalence*

AHardwareBuffer Usage	Vulkan Usage or Creation Flag
None	VK_IMAGE_USAGE_TRANSFER_SRC_BIT
None	VK_IMAGE_USAGE_TRANSFER_DST_BIT
AHARDWAREBUFFER_USAGE_GPU_SAMPLED_IMAGE	VK_IMAGE_USAGE_SAMPLED_BIT
AHARDWAREBUFFER_USAGE_GPU_SAMPLED_IMAGE	VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
AHARDWAREBUFFER_USAGE_GPU_COLOR_OUTPUT	VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
AHARDWAREBUFFER_USAGE_GPU_CUBE_MAP	VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT
AHARDWAREBUFFER_USAGE_GPU_MIPMAP_COMPLETE	None <sup>2</sup>
AHARDWAREBUFFER_USAGE_PROTECTED_CONTENT	VK_IMAGE_CREATE_PROTECTED_BIT
None	VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT
None	VK_IMAGE_CREATE_EXTENDED_USAGE_BIT

1

Vulkan does not differentiate between AHARDWAREBUFFER\_FORMAT\_R8G8B8A8\_UNORM and AHARDWAREBUFFER\_FORMAT\_R8G8B8X8\_UNORM: they both behave as VK\_FORMAT\_R8G8B8A8\_UNORM. After an external entity writes to a AHARDWAREBUFFER\_FORMAT\_R8G8B8X8\_UNORM Android hardware buffer, the values read by Vulkan from the X/A channel are undefined. To emulate the traditional behavior of the X channel during sampling or blending, applications **should** use VK\_COMPONENT\_SWIZZLE\_ONE in image view component mappings and VK\_BLEND\_FACTOR\_ONE in color blend factors. There is no way to avoid copying these undefined values when copying from such an image to another image or buffer.

2

The AHARDWAREBUFFER\_USAGE\_GPU\_MIPMAP\_COMPLETE flag does not correspond to a Vulkan image usage or creation flag. Instead, its presence indicates that the Android hardware buffer contains a complete mipmap chain, and its absence indicates that the Android hardware buffer contains

only a single mip level.

*Note*

When using `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` with Android hardware buffer images, applications **should** use `VkImageFormatListCreateInfoKHR` to inform the implementation which view formats will be used with the image. For some common sets of format, this allows some implementations to provide significantly better performance when accessing the image via Vulkan.



### Android Hardware Buffer Buffer Resources

Android hardware buffers with a format of `AHARDWAREBUFFER_FORMAT_BLOB` and usage that includes `AHARDWAREBUFFER_USAGE_GPU_DATA_BUFFER` **can** be used as the backing store for `VkBuffer` objects. Such Android hardware buffers have a size in bytes specified by their `width`; `height` and `layers` are both `1`.

Unlike images, buffer resources backed by Android hardware buffers do not require dedicated allocations.

Exported `AHardwareBuffer` objects that do not have dedicated images **must** have a format of `AHARDWAREBUFFER_FORMAT_BLOB`, usage **must** include `AHARDWAREBUFFER_USAGE_GPU_DATA_BUFFER`, `width` **must** equal the device memory allocation size, and `height` and `layers` **must** be `1`.

### 10.2.5. Peer Memory Features

*Peer memory* is memory that is allocated for a given physical device and then bound to a resource and accessed by a different physical device, in a logical device that represents multiple physical devices. Some ways of reading and writing peer memory **may** not be supported by a device.

To determine how peer memory **can** be accessed, call:

```
void vkGetDeviceGroupPeerMemoryFeatures(  
    VkDevice device,  
    uint32_t heapIndex,  
    uint32_t localDeviceIndex,  
    uint32_t remoteDeviceIndex,  
    VkPeerMemoryFeatureFlags* pPeerMemoryFeatures);
```

or the equivalent command

```
void vkGetDeviceGroupPeerMemoryFeaturesKHR(  
    VkDevice device,  
    uint32_t heapIndex,  
    uint32_t localDeviceIndex,  
    uint32_t remoteDeviceIndex,  
    VkPeerMemoryFeatureFlags* pPeerMemoryFeatures);
```

- `device` is the logical device that owns the memory.

- `heapIndex` is the index of the memory heap from which the memory is allocated.
- `localDeviceIndex` is the device index of the physical device that performs the memory access.
- `remoteDeviceIndex` is the device index of the physical device that the memory is allocated for.
- `pPeerMemoryFeatures` is a pointer to a `VkPeerMemoryFeatureFlags` bitmask indicating which types of memory accesses are supported for the combination of heap, local, and remote devices.

## Valid Usage

- `heapIndex` **must** be less than `memoryHeapCount`
- `localDeviceIndex` **must** be a valid device index
- `remoteDeviceIndex` **must** be a valid device index
- `localDeviceIndex` **must** not equal `remoteDeviceIndex`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pPeerMemoryFeatures` **must** be a valid pointer to a `VkPeerMemoryFeatureFlags` value

Bits which **may** be set in the value returned for `vkGetDeviceGroupPeerMemoryFeatures`::`pPeerMemoryFeatures`, indicating the supported peer memory features, are:

```
typedef enum VkPeerMemoryFeatureFlagBits {
    VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT = 0x00000001,
    VK_PEER_MEMORY_FEATURE_COPY_DST_BIT = 0x00000002,
    VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT = 0x00000004,
    VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT = 0x00000008,
    VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT_KHR = VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT,
    VK_PEER_MEMORY_FEATURE_COPY_DST_BIT_KHR = VK_PEER_MEMORY_FEATURE_COPY_DST_BIT,
    VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT_KHR =
VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT,
    VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT_KHR =
VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT,
    VK_PEER_MEMORY_FEATURE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPeerMemoryFeatureFlagBits;
```

or the equivalent

```
typedef VkPeerMemoryFeatureFlagBits VkPeerMemoryFeatureFlagBitsKHR;
```

- `VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT` specifies that the memory **can** be accessed as the source of a `vkCmdCopyBuffer`, `vkCmdCopyImage`, `vkCmdCopyBufferToImage`, or `vkCmdCopyImageToBuffer` command.

- `VK_PEER_MEMORY_FEATURE_COPY_DST_BIT` specifies that the memory **can** be accessed as the destination of a `vkCmdCopyBuffer`, `vkCmdCopyImage`, `vkCmdCopyBufferToImage`, or `vkCmdCopyImageToBuffer` command.
- `VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT` specifies that the memory **can** be read as any memory access type.
- `VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT` specifies that the memory **can** be written as any memory access type. Shader atomics are considered to be writes.

*Note*



The peer memory features of a memory heap also apply to any accesses that **may** be performed during [image layout transitions](#).

`VK_PEER_MEMORY_FEATURE_COPY_DST_BIT` **must** be supported for all host local heaps and for at least one device local heap.

If a device does not support a peer memory feature, it is still valid to use a resource that includes both local and peer memory bindings with the corresponding access type as long as only the local bindings are actually accessed. For example, an application doing split-frame rendering would use framebuffer attachments that include both local and peer memory bindings, but would scissor the rendering to only update local memory.

```
typedef VkFlags VkPeerMemoryFeatureFlags;
```

or the equivalent

```
typedef VkPeerMemoryFeatureFlags VkPeerMemoryFeatureFlagsKHR;
```

`VkPeerMemoryFeatureFlags` is a bitmask type for setting a mask of zero or more `VkPeerMemoryFeatureFlagBits`.

To query a 64-bit opaque capture address value from a memory object, call:

```
uint64_t vkGetDeviceMemoryOpaqueCaptureAddressKHR(
    VkDevice                           device,
    const VkDeviceMemoryOpaqueCaptureAddressInfoKHR* pInfo);
```

- `device` is the logical device that the memory object was allocated on.
- `pInfo` is a pointer to an instance of the `VkDeviceMemoryOpaqueCaptureAddressInfoKHR` structure specifying the memory object to retrieve an address for.

The 64-bit return value is an opaque address representing the start of `pInfo::memory`.

If the memory object was allocated with a non-zero value of `VkMemoryOpaqueCaptureAddressAllocateInfoKHR::opaqueCaptureAddress`, the return value **must** be the same address.

*Note*



The expected usage for these opaque addresses is only for trace capture/replay tools to store these addresses in a trace and subsequently specify them during replay.

## Valid Usage

- The `bufferDeviceAddress` feature **must** be enabled
- If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature **must** be enabled

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkDeviceMemoryOpaqueCaptureAddressInfoKHR` structure

The `VkDeviceMemoryOpaqueCaptureAddressInfoKHR` structure is defined as:

```
typedef struct VkDeviceMemoryOpaqueCaptureAddressInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
} VkDeviceMemoryOpaqueCaptureAddressInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memory` specifies the memory whose address is being queried.

## Valid Usage

- `memory` **must** have been allocated with `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO_KHR`
- `pNext` **must** be `NULL`
- `memory` **must** be a valid `VkDeviceMemory` handle

# Chapter 11. Resource Creation

Vulkan supports two primary resource types: *buffers* and *images*. Resources are views of memory with associated formatting and dimensionality. Buffers are essentially unformatted arrays of bytes whereas images contain format information, **can** be multidimensional and **may** have associated metadata.

## 11.1. Buffers

Buffers represent linear arrays of data which are used for various purposes by binding them to a graphics or compute pipeline via descriptor sets or via certain commands, or by directly specifying them as parameters to certain commands.

Buffers are represented by `VkBuffer` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBuffer)
```

To create buffers, call:

```
VkResult vkCreateBuffer(  
    VkDevice                                     device,  
    const VkBufferCreateInfo*                    pCreateInfo,  
    const VkAllocationCallbacks*                pAllocator,  
    VkBuffer*                                    pBuffer);
```

- `device` is the logical device that creates the buffer object.
- `pCreateInfo` is a pointer to a `VkBufferCreateInfo` structure containing parameters affecting creation of the buffer.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pBuffer` is a pointer to a `VkBuffer` handle in which the resulting buffer object is returned.

### Valid Usage

- If the `flags` member of `pCreateInfo` includes `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, creating this `VkBuffer` **must** not cause the total required sparse memory for all currently valid sparse resources on the device to exceed `VkPhysicalDeviceLimits::sparseAddressSpaceSize`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkBufferCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pBuffer` **must** be a valid pointer to a `VkBuffer` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR`

The `VkBufferCreateInfo` structure is defined as:

```
typedef struct VkBufferCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkBufferCreateFlags        flags;
    VkDeviceSize                size;
    VkBufferUsageFlags         usage;
    VkSharingMode              sharingMode;
    uint32_t                  queueFamilyIndexCount;
    const uint32_t*            pQueueFamilyIndices;
} VkBufferCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkBufferCreateFlagBits` specifying additional parameters of the buffer.
- `size` is the size in bytes of the buffer to be created.
- `usage` is a bitmask of `VkBufferUsageFlagBits` specifying allowed usages of the buffer.
- `sharingMode` is a `VkSharingMode` value specifying the sharing mode of the buffer when it will be accessed by multiple queue families.
- `queueFamilyIndexCount` is the number of entries in the `pQueueFamilyIndices` array.
- `pQueueFamilyIndices` is a list of queue families that will access this buffer (ignored if `sharingMode` is not `VK_SHARING_MODE_CONCURRENT`).

## Valid Usage

- `size` **must** be greater than `0`
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` **must** be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` **must** be greater than `1`
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` **must** be unique and **must** be less than `pQueueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create `device`
- If the `sparse bindings` feature is not enabled, `flags` **must** not contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`
- If the `sparse buffer residency` feature is not enabled, `flags` **must** not contain `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`
- If the `sparse aliased residency` feature is not enabled, `flags` **must** not contain `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`
- If `flags` contains `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`, it **must** also contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`
- If the `pNext` chain contains an instance of `VkExternalMemoryBufferCreateInfo`, its `handleTypes` member **must** only contain bits that are also in `VkExternalBufferProperties::externalMemoryProperties.compatibleHandleTypes`, as returned by `vkGetPhysicalDeviceExternalBufferProperties` with `pExternalBufferCreateInfo->handleType` equal to any one of the handle types specified in `VkExternalMemoryBufferCreateInfo::handleTypes`
- If the protected memory feature is not enabled, `flags` **must** not contain `VK_BUFFER_CREATE_PROTECTED_BIT`
- If any of the bits `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` are set, `VK_BUFFER_CREATE_PROTECTED_BIT` **must** not also be set
- If the `pNext` chain contains an instance of `VkDedicatedAllocationBufferCreateInfoNV`, and the `dedicatedAllocation` member of the chained structure is `VK_TRUE`, then `flags` **must** not include `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`
- If `VkBufferDeviceAddressCreateInfoEXT::deviceAddress` is not zero, `flags` **must** include `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`
- If `VkBufferOpaqueCaptureAddressCreateInfoKHR::opaqueCaptureAddress` is not zero, `flags` **must** include `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`
- If `flags` includes `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`, the `bufferDeviceAddressCaptureReplay` or `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT::bufferDeviceAddressCaptureReplay` feature **must** be enabled

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkBufferDeviceAddressCreateInfoEXT`, `VkBufferOpaqueCaptureAddressCreateInfoKHR`, `VkDedicatedAllocationBufferCreateInfoNV`, or `VkExternalMemoryBufferCreateInfo`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be a valid combination of `VkBufferCreateFlagBits` values
- `usage` **must** be a valid combination of `VkBufferUsageFlagBits` values
- `usage` **must** not be `0`
- `sharingMode` **must** be a valid `VkSharingMode` value

Bits which **can** be set in `VkBufferCreateInfo::usage`, specifying usage behavior of a buffer, are:

```
typedef enum VkBufferUsageFlagBits {
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_BUFFER_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000004,
    VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT = 0x00000008,
    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT = 0x00000010,
    VK_BUFFER_USAGE_STORAGE_BUFFER_BIT = 0x00000020,
    VK_BUFFER_USAGE_INDEX_BUFFER_BIT = 0x00000040,
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT = 0x00000080,
    VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT = 0x00000100,
    VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT = 0x00000800,
    VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT = 0x00001000,
    VK_BUFFER_USAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00000200,
    VK_BUFFER_USAGE_RAY_TRACING_BIT_NV = 0x00000400,
    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR = 0x00020000,
    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_EXT =
VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR,
    VK_BUFFER_USAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkBufferUsageFlagBits;
```

- `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` specifies that the buffer **can** be used as the source of a *transfer command* (see the definition of `VK_PIPELINE_STAGE_TRANSFER_BIT`).
- `VK_BUFFER_USAGE_TRANSFER_DST_BIT` specifies that the buffer **can** be used as the destination of a transfer command.
- `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` specifies that the buffer **can** be used to create a `VkBufferView` suitable for occupying a `VkDescriptorSet` slot of type `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`.
- `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` specifies that the buffer **can** be used to create a

`VkBufferView` suitable for occupying a `VkDescriptorSet` slot of type `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`.

- `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` specifies that the buffer **can** be used in a `VkDescriptorBufferInfo` suitable for occupying a `VkDescriptorSet` slot either of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`.
- `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT` specifies that the buffer **can** be used in a `VkDescriptorBufferInfo` suitable for occupying a `VkDescriptorSet` slot either of type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`.
- `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` specifies that the buffer is suitable for passing as the `buffer` parameter to `vkCmdBindIndexBuffer`.
- `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` specifies that the buffer is suitable for passing as an element of the `pBuffers` array to `vkCmdBindVertexBuffers`.
- `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` specifies that the buffer is suitable for passing as the `buffer` parameter to `vkCmdDrawIndirect`, `vkCmdDrawIndexedIndirect`, `vkCmdDrawMeshTasksIndirectNV`, `vkCmdDrawMeshTasksIndirectCountNV`, or `vkCmdDispatchIndirect`. It is also suitable for passing as the `buffer` member of `VkIndirectCommandsTokenNVX`, or `sequencesCountBuffer` or `sequencesIndexBuffer` member of `VkCmdProcessCommandsInfoNVX`
- `VK_BUFFER_USAGE_CONDITIONAL_RENDERING_BIT_EXT` specifies that the buffer is suitable for passing as the `buffer` parameter to `vkCmdBeginConditionalRenderingEXT`.
- `VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT` specifies that the buffer is suitable for using for binding as a transform feedback buffer with `vkCmdBindTransformFeedbackBuffersEXT`.
- `VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT` specifies that the buffer is suitable for using as a counter buffer with `vkCmdBeginTransformFeedbackEXT` and `vkCmdEndTransformFeedbackEXT`.
- `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` specifies that the buffer is suitable for use in `vkCmdTraceRaysNV` and `vkCmdBuildAccelerationStructureNV`.
- `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR` specifies that the buffer **can** be used to retrieve a buffer device address via `vkGetBufferDeviceAddressKHR` and use that address to access the buffer's memory from a shader.

```
typedef VkFlags VkBufferUsageFlags;
```

`VkBufferUsageFlags` is a bitmask type for setting a mask of zero or more `VkBufferUsageFlagBits`.

Bits which **can** be set in `VkBufferCreateInfo::flags`, specifying additional parameters of a buffer, are:

```

typedef enum VkBufferCreateFlagBits {
    VK_BUFFER_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_BUFFER_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
    VK_BUFFER_CREATE_PROTECTED_BIT = 0x00000008,
    VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR = 0x00000010,
    VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_EXT =
VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR,
    VK_BUFFER_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkBufferCreateFlagBits;

```

- **VK\_BUFFER\_CREATE\_SPARSE\_BINDING\_BIT** specifies that the buffer will be backed using sparse memory binding.
- **VK\_BUFFER\_CREATE\_SPARSE\_RESIDENCY\_BIT** specifies that the buffer **can** be partially backed using sparse memory binding. Buffers created with this flag **must** also be created with the **VK\_BUFFER\_CREATE\_SPARSE\_BINDING\_BIT** flag.
- **VK\_BUFFER\_CREATE\_SPARSE\_ALIASED\_BIT** specifies that the buffer will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another buffer (or another portion of the same buffer). Buffers created with this flag **must** also be created with the **VK\_BUFFER\_CREATE\_SPARSE\_BINDING\_BIT** flag.
- **VK\_BUFFER\_CREATE\_PROTECTED\_BIT** specifies that the buffer is a protected buffer.
- **VK\_BUFFER\_CREATE\_DEVICE\_ADDRESS\_CAPTURE\_REPLAY\_BIT\_KHR** specifies that the buffer's address **can** be saved and reused on a subsequent run (e.g. for trace capture and replay), see [VkBufferOpaqueCaptureAddressCreateInfoKHR](#) for more detail.

See [Sparse Resource Features](#) and [Physical Device Features](#) for details of the sparse memory features supported on a device.

```
typedef VkFlags VkBufferCreateFlags;
```

**VkBufferCreateFlags** is a bitmask type for setting a mask of zero or more **VkBufferCreateFlagBits**.

If the **pNext** chain includes a **VkDedicatedAllocationBufferCreateInfoNV** structure, then that structure includes an enable controlling whether the buffer will have a dedicated memory allocation bound to it.

The **VkDedicatedAllocationBufferCreateInfoNV** structure is defined as:

```

typedef struct VkDedicatedAllocationBufferCreateInfoNV {
    VkStructureType      sType;
    const void*        pNext;
    VkBool32            dedicatedAllocation;
} VkDedicatedAllocationBufferCreateInfoNV;

```

- **sType** is the type of this structure.

- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `dedicatedAllocation` specifies whether the buffer will have a dedicated allocation bound to it.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_BUFFER_CREATE_INFO_NV`

To define a set of external memory handle types that **may** be used as backing store for a buffer, add a `VkExternalMemoryBufferCreateInfo` structure to the `pNext` chain of the `VkBufferCreateInfo` structure. The `VkExternalMemoryBufferCreateInfo` structure is defined as:

```
typedef struct VkExternalMemoryBufferCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkExternalMemoryHandleTypeFlags handleTypes;
} VkExternalMemoryBufferCreateInfo;
```

or the equivalent

```
typedef VkExternalMemoryBufferCreateInfo VkExternalMemoryBufferCreateInfoKHR;
```

### Note



An instance of `VkExternalMemoryBufferCreateInfo` must be included in the creation parameters for a buffer that will be bound to memory that is either exported or imported.

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying one or more external memory handle types.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO`
- `handleTypes` must be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values

To request a specific device address for a buffer, add a `VkBufferOpaqueCaptureAddressCreateInfoKHR` structure to the `pNext` chain of the `VkBufferCreateInfo` structure. The `VkBufferOpaqueCaptureAddressCreateInfoKHR` structure is defined as:

```
typedef struct VkBufferOpaqueCaptureAddressCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint64_t opaqueCaptureAddress;
} VkBufferOpaqueCaptureAddressCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `opaqueCaptureAddress` is the opaque capture address requested for the buffer.

If `opaqueCaptureAddress` is zero, no specific address is requested.

If `opaqueCaptureAddress` is not zero, then it **should** be an address retrieved from `vkGetBufferOpaqueCaptureAddressKHR` for an identically created buffer on the same implementation.

If this structure is not present, it is as if `opaqueCaptureAddress` is zero.

Apps **should** avoid creating buffers with app-provided addresses and implementation-provided addresses in the same process, to reduce the likelihood of `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR` errors.

*Note*

The expected usage for this is that a trace capture/replay tool will add the `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR` flag to all buffers that use `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR`, and during capture will save the queried opaque device addresses in the trace. During replay, the buffers will be created specifying the original address so any address values stored in the trace data will remain valid.

Implementations are expected to separate such buffers in the GPU address space so normal allocations will avoid using these addresses. Apps/tools should avoid mixing app-provided and implementation-provided addresses for buffers created with `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`, to avoid address space allocation conflicts.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO_KHR`

Alternatively, to request a specific device address for a buffer, add a `VkBufferDeviceAddressCreateInfoEXT` structure to the `pNext` chain of the `VkBufferCreateInfo` structure. The `VkBufferDeviceAddressCreateInfoEXT` structure is defined as:

```
typedef struct VkBufferDeviceAddressCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDeviceAddress deviceAddress;
} VkBufferDeviceAddressCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `deviceAddress` is the device address requested for the buffer.

If `deviceAddress` is zero, no specific address is requested.

If `deviceAddress` is not zero, then it **must** be an address retrieved from an identically created buffer on the same implementation. The buffer **must** also be bound to an identically created `VkDeviceMemory` object.

If this structure is not present, it is as if `deviceAddress` is zero.

Apps **should** avoid creating buffers with app-provided addresses and implementation-provided addresses in the same process, to reduce the likelihood of `VK_ERROR_INVALID_DEVICE_ADDRESS_EXT` errors.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_CREATE_INFO_EXT`

To destroy a buffer, call:

```
void vkDestroyBuffer(
    VkDevice device,
    VkBuffer buffer,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the buffer.
- `buffer` is the buffer to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `buffer`, either directly or via a `VkBufferView`, **must** have completed execution
- If `VkAllocationCallbacks` were provided when `buffer` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `buffer` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `buffer` is not `VK_NULL_HANDLE`, `buffer` **must** be a valid `VkBuffer` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `buffer` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `buffer` **must** be externally synchronized

## 11.2. Buffer Views

A *buffer view* represents a contiguous range of a buffer and a specific format to be used to interpret the data. Buffer views are used to enable shaders to access buffer contents interpreted as formatted data. In order to create a valid buffer view, the buffer **must** have been created with at least one of the following usage flags:

- `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`
- `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`

Buffer views are represented by `VkBufferView` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBufferView)
```

To create a buffer view, call:

```

VkResult vkCreateBufferView(
    VkDevice                                     device,
    const VkBufferViewCreateInfo*                 pCreateInfo,
    const VkAllocationCallbacks*                 pAllocator,
    VkBufferView*                                pView);

```

- `device` is the logical device that creates the buffer view.
- `pCreateInfo` is a pointer to a `VkBufferViewCreateInfo` structure containing parameters to be used to create the buffer.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pView` is a pointer to a `VkBufferView` handle in which the resulting buffer view object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkBufferViewCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pView` **must** be a valid pointer to a `VkBufferView` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkBufferViewCreateInfo` structure is defined as:

```

typedef struct VkBufferViewCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkBufferViewCreateFlags   flags;
    VkBuffer                  buffer;
    VkFormat                  format;
    VkDeviceSize               offset;
    VkDeviceSize               range;
} VkBufferViewCreateInfo;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `buffer` is a `VkBuffer` on which the view will be created.
- `format` is a `VkFormat` describing the format of the data elements in the buffer.
- `offset` is an offset in bytes from the base address of the buffer. Accesses to the buffer view from shaders use addressing that is relative to this starting offset.
- `range` is a size in bytes of the buffer view. If `range` is equal to `VK_WHOLE_SIZE`, the range from `offset` to the end of the buffer is used. If `VK_WHOLE_SIZE` is used and the remaining size of the buffer is not a multiple of the `texel block size` of `format`, the nearest smaller multiple is used.

## Valid Usage

- `offset` **must** be less than the size of `buffer`
- If `range` is not equal to `VK_WHOLE_SIZE`, `range` **must** be greater than `0`
- If `range` is not equal to `VK_WHOLE_SIZE`, `range` **must** be an integer multiple of the texel block size of `format`
- If `range` is not equal to `VK_WHOLE_SIZE`, `range` divided by the texel block size of `format`, multiplied by the number of texels per texel block for that format (as defined in the [Compatible Formats](#) table), **must** be less than or equal to `VkPhysicalDeviceLimits::maxTexelBufferElements`
- If `range` is not equal to `VK_WHOLE_SIZE`, the sum of `offset` and `range` **must** be less than or equal to the size of `buffer`
- `buffer` **must** have been created with a `usage` value containing at least one of `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`
- If `buffer` was created with `usage` containing `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`, `format` **must** be supported for uniform texel buffers, as specified by the `VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT` flag in `VkFormatProperties::bufferFeatures` returned by `vkGetPhysicalDeviceFormatProperties`
- If `buffer` was created with `usage` containing `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, `format` **must** be supported for storage texel buffers, as specified by the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` flag in `VkFormatProperties::bufferFeatures` returned by `vkGetPhysicalDeviceFormatProperties`
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- If the `texelBufferAlignment` feature is not enabled, `offset` **must** be a multiple of `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`
- If the `texelBufferAlignment` feature is enabled and if `buffer` was created with `usage` containing `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, `offset` **must** be a multiple of the lesser of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::storageTexelBufferOffsetAlignmentBytes` or, if `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::storageTexelBufferOffsetSingleTexelAlignment` is `VK_TRUE`, the size of a texel of the requested `format`. If the size of a texel is a multiple of three bytes, then the size of a single component of `format` is used instead
- If the `texelBufferAlignment` feature is enabled and if `buffer` was created with `usage` containing `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`, `offset` **must** be a multiple of the lesser of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::uniformTexelBufferOffsetAlignmentBytes` or, if `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::uniformTexelBufferOffsetSingleTexelAlignment` is `VK_TRUE`, the size of a texel of the requested `format`. If the size of a texel is a multiple of three bytes, then the size of a single component of `format` is used instead

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `buffer` **must** be a valid `VkBuffer` handle
- `format` **must** be a valid `VkFormat` value

```
typedef VkFlags VkBufferViewCreateFlags;
```

`VkBufferViewCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To destroy a buffer view, call:

```
void vkDestroyBufferView(  
    VkDevice device,  
    VkBufferView bufferView,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the buffer view.
- `bufferView` is the buffer view to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `bufferView` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `bufferView` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `bufferView` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `bufferView` is not `VK_NULL_HANDLE`, `bufferView` **must** be a valid `VkBufferView` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `bufferView` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `bufferView` **must** be externally synchronized

### 11.3. Images

Images represent multidimensional - up to 3 - arrays of data which **can** be used for various purposes (e.g. attachments, textures), by binding them to a graphics or compute pipeline via descriptor sets, or by directly specifying them as parameters to certain commands.

Images are represented by `VkImage` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImage)
```

To create images, call:

```
VkResult vkCreateImage(  
    VkDevice                                     device,  
    const VkImageCreateInfo*                      pCreateInfo,  
    const VkAllocationCallbacks*                  pAllocator,  
    VkImage*                                     pImage);
```

- `device` is the logical device that creates the image.
- `pCreateInfo` is a pointer to a `VkImageCreateInfo` structure containing parameters to be used to create the image.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pImage` is a pointer to a `VkImage` handle in which the resulting image object is returned.

## Valid Usage

- If the `flags` member of `pCreateInfo` includes `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, creating this `VkImage` **must** not cause the total required sparse memory for all currently valid sparse resources on the device to exceed `VkPhysicalDeviceLimits::sparseAddressSpaceSize`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkImageCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pImage` **must** be a valid pointer to a `VkImage` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkImageCreateInfo` structure is defined as:

```
typedef struct VkImageCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkImageCreateFlags        flags;
    VkImageType               imageType;
    VkFormat                  format;
    VkExtent3D                extent;
    uint32_t                  mipLevels;
    uint32_t                  arrayLayers;
    VkSampleCountFlagBits     samples;
    VkImageTiling              tiling;
    VkImageUsageFlags         usage;
    VkSharingMode              sharingMode;
    uint32_t                  queueFamilyIndexCount;
    const uint32_t*            pQueueFamilyIndices;
    VkImageLayout              initialLayout;
} VkImageCreateInfo;
```

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkImageCreateFlagBits` describing additional parameters of the image.
- `imageType` is a `VkImageType` value specifying the basic dimensionality of the image. Layers in array textures do not count as a dimension for the purposes of the image type.
- `format` is a `VkFormat` describing the format and type of the texel blocks that will be contained in the image.
- `extent` is a `VkExtent3D` describing the number of data elements in each dimension of the base level.
- `mipLevels` describes the number of levels of detail available for minified sampling of the image.
- `arrayLayers` is the number of layers in the image.
- `samples` is a `VkSampleCountFlagBits` specifying the number of samples per texel.
- `tiling` is a `VkImageTiling` value specifying the tiling arrangement of the texel blocks in memory.
- `usage` is a bitmask of `VkImageUsageFlagBits` describing the intended usage of the image.
- `sharingMode` is a `VkSharingMode` value specifying the sharing mode of the image when it will be accessed by multiple queue families.
- `queueFamilyIndexCount` is the number of entries in the `pQueueFamilyIndices` array.
- `pQueueFamilyIndices` is a list of queue families that will access this image (ignored if `sharingMode` is not `VK_SHARING_MODE_CONCURRENT`).
- `initialLayout` is a `VkImageLayout` value specifying the initial `VkImageLayout` of all image subresources of the image. See [Image Layouts](#).

Images created with `tiling` equal to `VK_IMAGE_TILING_LINEAR` have further restrictions on their limits and capabilities compared to images created with `tiling` equal to `VK_IMAGE_TILING_OPTIMAL`. Creation of images with tiling `VK_IMAGE_TILING_LINEAR` **may** not be supported unless other parameters meet all of the constraints:

- `imageType` is `VK_IMAGE_TYPE_2D`
- `format` is not a depth/stencil format
- `mipLevels` is 1
- `arrayLayers` is 1
- `samples` is `VK_SAMPLE_COUNT_1_BIT`
- `usage` only includes `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` and/or `VK_IMAGE_USAGE_TRANSFER_DST_BIT`

Images created with a `format` from one of those listed in [Formats requiring sampler Y'C<sub>B</sub>C<sub>R</sub> conversion for VK\\_IMAGE\\_ASPECT\\_COLOR\\_BIT image views](#) have further restrictions on their limits and capabilities compared to images created with other formats. Creation of images with a format requiring `Y'CBCR conversion` **may** not be supported unless other parameters meet all of the constraints:

- `imageType` is `VK_IMAGE_TYPE_2D`
- `mipLevels` is 1

- `arrayLayers` is 1
- `samples` is `VK_SAMPLE_COUNT_1_BIT`

Implementations **may** support additional limits and capabilities beyond those listed above.

To determine the set of valid `usage` bits for a given format, call [vkGetPhysicalDeviceFormatProperties](#).

If the size of the resultant image would exceed `maxResourceSize`, then [vkCreateImage](#) **must** fail and return `VK_ERROR_OUT_OF_DEVICE_MEMORY`. This failure **may** occur even when all image creation parameters satisfy their valid usage requirements.

*Note*

For images created without `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a `usage` bit is valid if it is supported for the format the image is created with.



For images created with `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a `usage` bit is valid if it is supported for at least one of the formats a [VkImageView](#) created from the image **can** have (see [Image Views](#) for more detail).

## Image Creation Limits

Valid values for some image creation parameters are limited by a numerical upper bound or by inclusion in a bitset. For example, `VkImageCreateInfo::arrayLayers` is limited by `imageCreateMaxArrayLayers`, defined below; and `VkImageCreateInfo::samples` is limited by `imageCreateSampleCounts`, also defined below.

Several limiting values are defined below, as well as assisting values from which the limiting values are derived. The limiting values are referenced by the relevant valid usage statements of `VkImageCreateInfo`.

- Let `uint64_t imageCreateDrmFormatModifiers[]` be the set of Linux DRM format modifiers that the resultant image **may** have.
  - If `tiling` is not `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `imageCreateDrmFormatModifiers` is empty.
  - If `VkImageCreateInfo::pNext` contains `VkImageDrmFormatModifierExplicitCreateInfoEXT`, then `imageCreateDrmFormatModifiers` contains exactly one modifier, `VkImageDrmFormatModifierExplicitCreateInfoEXT::drmFormatModifier`.
  - If `VkImageCreateInfo::pNext` contains `VkImageDrmFormatModifierListCreateInfoEXT`, then `imageCreateDrmFormatModifiers` contains the exactly the modifiers in `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`.
- Let `VkBool32 imageCreateMaybeLinear` indicate if the resultant image may be `linear`.
  - If `tiling` is `VK_IMAGE_TILING_LINEAR`, then `imageCreateMaybeLinear` is `true`.
  - If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, then `imageCreateMaybeLinear` is `false`.
  - If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `imageCreateMaybeLinear` is `true` if and only if `imageCreateDrmFormatModifiers` contains `DRM_FORMAT_MOD_LINEAR`.
- Let `VkFormatFeatureFlags imageCreateFormatFeatures` be the set of format features available during image creation.
  - If `tiling` is `VK_IMAGE_TILING_LINEAR`, then `imageCreateFormatFeatures` is the value of `VkImageFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.
  - If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, and if the `pNext` chain contains no instance of `VkExternalFormatANDROID` with non-zero `externalFormat`, then `imageCreateFormatFeatures` is value of `VkImageFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.
  - If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, and if the `pNext` chain contains an instance of `VkExternalFormatANDROID` with non-zero `externalFormat`, then `imageCreateFormatFeatures` is the value of `VkAndroidHardwareBufferFormatPropertiesANDROID::formatFeatures` obtained by `vkGetAndroidHardwareBufferPropertiesANDROID` with a matching `externalFormat`

value.

- If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the value of `imageCreateFormatFeatures` is found by calling `vkGetPhysicalDeviceFormatProperties2` with `VkImageFormatProperties::format` equal to `VkImageCreateInfo::format` and with `VkDrmFormatModifierPropertiesListEXT` chained into `VkImageFormatProperties2`; by collecting all members of the returned array `VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties` whose `drmFormatModifier` belongs to `imageCreateDrmFormatModifiers`; and by taking the bitwise intersection, over the collected array members, of `drmFormatModifierTilingFeatures`. (The resultant `imageCreateFormatFeatures` **may** be empty).
- Let `VkImageFormatProperties2 imageCreateImageFormatPropertiesList[]` be defined as follows.
  - If `VkImageCreateInfo::pNext` contains no instance of `VkExternalFormatANDROID` with non-zero `externalFormat`, then `imageCreateImageFormatPropertiesList` is the list of structures obtained by calling `vkGetPhysicalDeviceImageFormatProperties2`, possibly multiple times, as follows:
    - The parameters `VkPhysicalDeviceImageFormatInfo2::format`, `imageType`, `tiling`, `usage`, and `flags` **must** be equal to those in `VkImageCreateInfo`.
    - If `VkImageCreateInfo::pNext` contains an instance of `VkExternalMemoryImageCreateInfo` where `handleTypes` is not `0`, then `VkPhysicalDeviceImageFormatInfo2::pNext` **must** contain an instance of `VkPhysicalDeviceExternalImageFormatInfo` where `handleType` is not `0`; and `vkGetPhysicalDeviceImageFormatProperties2` **must** be called for each handle type in `VkExternalMemoryImageCreateInfo::handleTypes`, successively setting `VkPhysicalDeviceExternalImageFormatInfo::handleType` on each call.
    - If `VkImageCreateInfo::pNext` contains no instance of `VkExternalMemoryImageCreateInfo` or contains an instance where `handleTypes` is `0`, then `VkPhysicalDeviceImageFormatInfo2::pNext` **must** either contain no instance of `VkPhysicalDeviceExternalImageFormatInfo` or contain an instance where `handleType` is `0`.
    - If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `VkPhysicalDeviceImageFormatInfo2::pNext` **must** contain an instance of `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` where `sharingMode` is equal to `VkImageCreateInfo::sharingMode`; and, if `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `queueFamilyIndexCount` and `pQueueFamilyIndices` **must** be equal to those in `VkImageCreateInfo`; and, if `flags` contains `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, then the instance of `VkImageFormatListCreateInfoKHR` in the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` **must** be equivalent to the one in the `pNext` chain of `VkImageCreateInfo`; and `vkGetPhysicalDeviceImageFormatProperties2` **must** be called for each modifier in `imageCreateDrmFormatModifiers`, successively setting `VkPhysicalDeviceImageDrmFormatModifierInfoEXT::drmFormatModifier` on each call.
    - If `tiling` is not `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `VkPhysicalDeviceImageFormatInfo2::pNext` **must** contain no instance of `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`.

- If any call to `vkGetPhysicalDeviceImageFormatProperties2` returns an error, then `imageCreateInfoPropertiesList` is defined to be the empty list.
- If `VkImageCreateInfo::pNext` contains an instance of `VkExternalFormatANDROID` with non-zero `externalFormat`, then `imageCreateInfoPropertiesList` contains a single element where:
  - `VkImageFormatProperties::maxMipLevels` is  $\log_2(\max(extent.width, extent.height, extent.depth)) + 1$ .
  - `VkImageFormatProperties::maxArrayLayers` is `VkPhysicalDeviceLimits::maxImageArrayLayers`.
  - Each component of `VkImageFormatProperties::maxExtent` is `VkPhysicalDeviceLimits::maxImageDimension2D`.
  - `VkImageFormatProperties::sampleCounts` contains exactly `VK_SAMPLE_COUNT_1_BIT`.
- Let `uint32_t imageCreateMaxMipLevels` be the minimum value of `VkImageFormatProperties::maxMipLevels` in `imageCreateInfoPropertiesList`. The value is undefined if `imageCreateInfoPropertiesList` is empty.
- Let `uint32_t imageCreateMaxArrayLayers` be the minimum value of `VkImageFormatProperties::maxArrayLayers` in `imageCreateInfoPropertiesList`. The value is undefined if `imageCreateInfoPropertiesList` is empty.
- Let `VkExtent3D imageCreateMaxExtent` be the component-wise minimum over all `VkImageFormatProperties::maxExtent` values in `imageCreateInfoPropertiesList`. The value is undefined if `imageCreateInfoPropertiesList` is empty.
- Let `VkSampleCountFlags imageCreateSampleCounts` be the intersection of each `VkImageFormatProperties::sampleCounts` in `imageCreateInfoPropertiesList`. The value is undefined if `imageCreateInfoPropertiesList` is empty.

## Valid Usage

- Each of the following values (as described in [Image Creation Limits](#)) **must** not be undefined `imageCreateMaxMipLevels`, `imageCreateMaxArrayLayers`, `imageCreateMaxExtent`, and `imageCreateSampleCounts`.
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` **must** be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` **must** be greater than 1
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` **must** be unique and **must** be less than `pQueueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create `device`
- If the `pNext` chain contains an instance of `VkExternalFormatANDROID`, and its member `externalFormat` is non-zero the `format` **must** be `VK_FORMAT_UNDEFINED`.
- If the `pNext` chain does not contain an instance of `VkExternalFormatANDROID`, or does and its member `externalFormat` is 0 the `format` **must** not be `VK_FORMAT_UNDEFINED`.
- `extent::width` **must** be greater than 0.
- `extent::height` **must** be greater than 0.
- `extent::depth` **must** be greater than 0.
- `mipLevels` **must** be greater than 0
- `arrayLayers` **must** be greater than 0
- If `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `imageType` **must** be `VK_IMAGE_TYPE_2D`
- If `flags` contains `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `imageType` **must** be `VK_IMAGE_TYPE_2D`
- If `flags` contains `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`, `imageType` **must** be `VK_IMAGE_TYPE_3D`
- `extent.width` **must** be less than or equal to `imageCreateMaxExtent.width` (as defined in [Image Creation Limits](#)).
- `extent.height` **must** be less than or equal to `imageCreateMaxExtent.height` (as defined in [Image Creation Limits](#)).
- `extent.depth` **must** be less than or equal to `imageCreateMaxExtent.depth` (as defined in [Image Creation Limits](#)).
- If `imageType` is `VK_IMAGE_TYPE_2D` and `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `extent.width` and `extent.height` **must** be equal and `arrayLayers` **must** be greater than or equal to 6
- If `imageType` is `VK_IMAGE_TYPE_1D`, both `extent.height` and `extent.depth` **must** be 1
- If `imageType` is `VK_IMAGE_TYPE_2D`, `extent.depth` **must** be 1
- `mipLevels` **must** be less than or equal to the number of levels in the complete mipmap

chain based on `extent.width`, `extent.height`, and `extent.depth`.

- `mipLevels` **must** be less than or equal to `imageCreateMaxMipLevels` (as defined in [Image Creation Limits](#)).
- `arrayLayers` **must** be less than or equal to `imageCreateMaxArrayLayers` (as defined in [Image Creation Limits](#)).
- If `imageType` is `VK_IMAGE_TYPE_3D`, `arrayLayers` **must** be 1.
- If `samples` is not `VK_SAMPLE_COUNT_1_BIT`, then `imageType` **must** be `VK_IMAGE_TYPE_2D`, `flags` **must** not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `mipLevels` **must** be equal to 1, and `imageCreateMaybeLinear` (as defined in [Image Creation Limits](#)) **must** be `false`,
- If `samples` is not `VK_SAMPLE_COUNT_1_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`
- If `usage` includes `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, then bits other than `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` **must** not be set
- If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferWidth`
- If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferHeight`
- If `usage` includes `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `extent.width` **must** be less than or equal to  $\lceil \frac{\text{maxFramebufferWidth}}{\text{minFragmentDensityTexelSize}_{\text{width}}} \rceil$
- If `usage` includes `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `extent.height` **must** be less than or equal to  $\lceil \frac{\text{maxFramebufferHeight}}{\text{minFragmentDensityTexelSize}_{\text{height}}} \rceil$
- If `usage` includes `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, `usage` **must** also contain at least one of `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.
- `samples` **must** be a bit value that is set in `imageCreateSampleCounts` (as defined in [Image Creation Limits](#)).
- If the `multisampled storage images` feature is not enabled, and `usage` contains `VK_IMAGE_USAGE_STORAGE_BIT`, `samples` **must** be `VK_SAMPLE_COUNT_1_BIT`
- If the `sparse bindings` feature is not enabled, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`
- If the `sparse aliased residency` feature is not enabled, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`
- If `imageType` is `VK_IMAGE_TYPE_1D`, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`
- If the `sparse residency for 2D images` feature is not enabled, and `imageType` is `VK_IMAGE_TYPE_2D`, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`
- If the `sparse residency for 3D images` feature is not enabled, and `imageType` is

`VK_IMAGE_TYPE_3D`, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`

- If the `sparse residency for images with 2 samples` feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_2_BIT`, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`
- If the `sparse residency for images with 4 samples` feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_4_BIT`, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`
- If the `sparse residency for images with 8 samples` feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_8_BIT`, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`
- If the `sparse residency for images with 16 samples` feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_16_BIT`, `flags` **must** not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`
- If `flags` contains `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`, it **must** also contain `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`
- If any of the bits `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` are set, `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` **must** not also be set
- If the protected memory feature is not enabled, `flags` **must** not contain `VK_IMAGE_CREATE_PROTECTED_BIT`.
- If any of the bits `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` are set, `VK_IMAGE_CREATE_PROTECTED_BIT` **must** not also be set.
- If the `pNext` chain contains an instance of `VkExternalMemoryImageCreateInfoNV`, it **must** not contain an instance of `VkExternalMemoryImageCreateInfo`.
- If the `pNext` chain contains an instance of `VkExternalMemoryImageCreateInfo`, its `handleTypes` member **must** only contain bits that are also in `VkExternalImageFormatProperties::externalMemoryProperties.compatibleHandleTypes`, as returned by `vkGetPhysicalDeviceImageFormatProperties2` with `format`, `imageType`, `tiling`, `usage`, and `flags` equal to those in this structure, and with an instance of `VkPhysicalDeviceExternalImageFormatInfo` in the `pNext` chain, with a `handleType` equal to any one of the handle types specified in `VkExternalMemoryImageCreateInfo::handleTypes`
- If the `pNext` chain contains an instance of `VkExternalMemoryImageCreateInfoNV`, its `handleTypes` member **must** only contain bits that are also in `VkExternalImageFormatPropertiesNV::externalMemoryProperties.compatibleHandleTypes`, as returned by `vkGetPhysicalDeviceExternalImageFormatPropertiesNV` with `format`, `imageType`, `tiling`, `usage`, and `flags` equal to those in this structure, and with `externalHandleType` equal to any one of the handle types specified in `VkExternalMemoryImageCreateInfoNV::handleTypes`
- If the logical device was created with `VkDeviceGroupDeviceCreateInfo::physicalDeviceCount` equal to 1, `flags` **must** not contain `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`
- If `flags` contains `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`, then `mipLevels` **must**

be one, `arrayLayers` **must** be one, `imageType` **must** be `VK_IMAGE_TYPE_2D`, and `imageCreateMaybeLinear` (as defined in [Image Creation Limits](#)) **must** be `false`.

- If `flags` contains `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`, then `format` **must** be a block-compressed image format, an ETC compressed image format, or an ASTC compressed image format.
- If `flags` contains `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`, then `flags` **must** also contain `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`.
- `initialLayout` **must** be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.
- If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` or `VkExternalMemoryImageCreateInfoNV` structure whose `handleTypes` member is not `0`, `initialLayout` **must** be `VK_IMAGE_LAYOUT_UNDEFINED`
- If the image `format` is one of those listed in [Formats requiring sampler Y'CbCr conversion for VK\\_IMAGE\\_ASPECT\\_COLOR\\_BIT image views](#), then `mipLevels` **must** be 1
- If the image `format` is one of those listed in [Formats requiring sampler Y'CbCr conversion for VK\\_IMAGE\\_ASPECT\\_COLOR\\_BIT image views](#), `samples` must be `VK_SAMPLE_COUNT_1_BIT`
- If the image `format` is one of those listed in [Formats requiring sampler Y'CbCr conversion for VK\\_IMAGE\\_ASPECT\\_COLOR\\_BIT image views](#), `imageType` **must** be `VK_IMAGE_TYPE_2D`
- If the image `format` is one of those listed in [Formats requiring sampler Y'CbCr conversion for VK\\_IMAGE\\_ASPECT\\_COLOR\\_BIT image views](#), and the `ycbcrImageArrays` feature is not enabled, `arrayLayers` **must** be 1
- If `format` is a *multi-planar* format, and if `imageCreateFormatFeatures` (as defined in [Image Creation Limits](#)) does not contain `VK_FORMAT_FEATURE_DISJOINT_BIT`, then `flags` **must** not contain `VK_IMAGE_CREATE_DISJOINT_BIT`.
- If `format` is not a *multi-planar* format, and `flags` does not include `VK_IMAGE_CREATE_ALIAS_BIT`, `flags` **must** not contain `VK_IMAGE_CREATE_DISJOINT_BIT`
- If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the `pNext` chain **must** contain exactly one of `VkImageDrmFormatModifierListCreateInfoEXT` or `VkImageDrmFormatModifierExplicitCreateInfoEXT`.
- If the `pNext` chain contains `VkImageDrmFormatModifierListCreateInfoEXT` or `VkImageDrmFormatModifierExplicitCreateInfoEXT`, then `tiling` **must** be `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`.
- If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and `flags` contains `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, then the `pNext` chain **must** contain `VkImageFormatListCreateInfoKHR` with non-zero `viewFormatCount`.
- If `flags` contains `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` `format` **must** be a depth or depth/stencil format
- If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` structure whose `handleTypes` member includes `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `imageType` **must** be `VK_IMAGE_TYPE_2D`.
- If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` structure whose `handleTypes` member includes

`VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `mipLevels` **must** either be 1 or equal to the number of levels in the complete mipmap chain based on `extent.width`, `extent.height`, and `extent.depth`.

- If the `pNext` chain includes a `VkExternalFormatANDROID` structure whose `externalFormat` member is not 0, `flags` **must** not include `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`.
- If the `pNext` chain includes a `VkExternalFormatANDROID` structure whose `externalFormat` member is not 0, `usage` **must** not include any usages except `VK_IMAGE_USAGE_SAMPLED_BIT`.
- If the `pNext` chain includes a `VkExternalFormatANDROID` structure whose `externalFormat` member is not 0, `tiling` **must** be `VK_IMAGE_TILING_OPTIMAL`.
- If `format` is a depth-stencil format, `usage` **includes** `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and the `pNext` chain contains an instance of `VkImageStencilUsageCreateInfoEXT`, then `VkImageStencilUsageCreateInfoEXT::stencilUsage` member **must** also include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If `format` is a depth-stencil format, `usage` does not include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and the `pNext` chain contains an instance of `VkImageStencilUsageCreateInfoEXT`, then `VkImageStencilUsageCreateInfoEXT::stencilUsage` member **must** also not include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If `format` is a depth-stencil format, `usage` **includes** `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, and the `pNext` chain contains an instance of `VkImageStencilUsageCreateInfoEXT`, then `VkImageStencilUsageCreateInfoEXT::stencilUsage` member **must** also include `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`
- If `format` is a depth-stencil format, `usage` does not include `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, and the `pNext` chain contains an instance of `VkImageStencilUsageCreateInfoEXT`, then `VkImageStencilUsageCreateInfoEXT::stencilUsage` member **must** also not include `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`
- If `Format` is a depth-stencil format and the `pNext` chain contains an instance of `VkImageStencilUsageCreateInfoEXT` with its `stencilUsage` member including `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferWidth`
- If `format` is a depth-stencil format and the `pNext` chain contains an instance of `VkImageStencilUsageCreateInfoEXT` with its `stencilUsage` member including `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferHeight`
- If the `multisampled storage images` feature is not enabled, `format` is a depth-stencil format and the `pNext` chain contains an instance of `VkImageStencilUsageCreateInfoEXT` with its `stencilUsage` including `VK_IMAGE_USAGE_STORAGE_BIT`, `samples` **must** be `VK_SAMPLE_COUNT_1_BIT`
- If `flags` contains `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV`, `imageType` **must** be `VK_IMAGE_TYPE_2D` or `VK_IMAGE_TYPE_3D`
- If `flags` contains `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV`, it **must** not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` and the `format` **must** not be a depth/stencil format
- If `flags` contains `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` and `imageType` is `VK_IMAGE_TYPE_2D`,

`extent::width` and `extent::height` **must** be greater than 1

- If `flags` contains `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` and `imageType` is `VK_IMAGE_TYPE_3D`, `extent::width`, `extent::height`, and `extent::depth` **must** be greater than 1
- If `usage` includes `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`, `imageType` **must** be `VK_IMAGE_TYPE_2D`.
- If `usage` includes `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`, `samples` **must** be `VK_SAMPLE_COUNT_1_BIT`.
- If `usage` includes `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`, `tiling` **must** be `VK_IMAGE_TILING_OPTIMAL`.
- If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `tiling` **must** be `VK_IMAGE_TILING_OPTIMAL`
- If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `imageType` **must** be `VK_IMAGE_TYPE_2D`
- If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `flags` **must** not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`
- If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `mipLevels` **must** be 1

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkDedicatedAllocationImageCreateInfoNV`, `VkExternalFormatANDROID`, `VkExternalMemoryImageCreateInfo`, `VkExternalMemoryImageCreateInfoNV`, `VkImageDrmFormatModifierExplicitCreateInfoEXT`, `VkImageDrmFormatModifierListCreateInfoEXT`, `VkImageFormatListCreateInfoKHR`, `VkImageStencilUsageCreateInfoEXT`, or `VkImageSwapchainCreateInfoKHR`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be a valid combination of `VkImageCreateFlagBits` values
- `imageType` **must** be a valid `VkImageType` value
- `format` **must** be a valid `VkFormat` value
- `samples` **must** be a valid `VkSampleCountFlagBits` value
- `tiling` **must** be a valid `VkImageTiling` value
- `usage` **must** be a valid combination of `VkImageUsageFlagBits` values
- `usage` **must** not be 0
- `sharingMode` **must** be a valid `VkSharingMode` value
- `initialLayout` **must** be a valid `VkImageLayout` value

If the `pNext` chain of `VkImageCreateInfo` includes a `VkImageStencilUsageCreateInfoEXT` structure, then that structure includes the usage flags specific to the stencil aspect of the image for an image

with a depth-stencil format.

The `VkImageStencilUsageCreateInfoEXT` structure is defined as:

```
typedef struct VkImageStencilUsageCreateInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkImageUsageFlags  stencilUsage;
} VkImageStencilUsageCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `stencilUsage` is a bitmask of `VkImageUsageFlagBits` describing the intended usage of the stencil aspect of the image.

This structure specifies image usages which only apply to the stencil aspect of a depth/stencil format image. When this structure is included in the `pNext` chain of `VkImageCreateInfo`, the stencil aspect of the image **must** only be used as specified by `stencilUsage`. When this structure is not included in the `pNext` chain of `VkImageCreateInfo`, the stencil aspect of an image **must** only be used as specified `VkImageCreateInfo::usage`. Use of other aspects of an image are unaffected by this structure.

This structure **can** also be included in the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` to query additional capabilities specific to image creation parameter combinations including a separate set of usage flags for the stencil aspect of the image using `vkGetPhysicalDeviceImageFormatProperties2`. When this structure is not present in the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` then the implicit value of `stencilUsage` matches that of `VkPhysicalDeviceImageFormatInfo2::usage`.

## Valid Usage

- If `stencilUsage` includes `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, it **must** not include bits other than `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO_EXT`
- `stencilUsage` **must** be a valid combination of `VkImageUsageFlagBits` values
- `stencilUsage` **must** not be `0`

If the `pNext` chain includes a `VkDedicatedAllocationImageCreateInfoNV` structure, then that structure includes an enable controlling whether the image will have a dedicated memory allocation bound to it.

The `VkDedicatedAllocationImageCreateInfoNV` structure is defined as:

```
typedef struct VkDedicatedAllocationImageCreateInfoNV {
    VkStructureType    sType;
    const void*        pNext;
    VkBool32           dedicatedAllocation;
} VkDedicatedAllocationImageCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `dedicatedAllocation` specifies whether the image will have a dedicated allocation bound to it.

*Note*



Using a dedicated allocation for color and depth/stencil attachments or other large images **may** improve performance on some devices.

## Valid Usage

- If `dedicatedAllocation` is `VK_TRUE`, `VkImageCreateInfo::flags` **must** not include `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV`

To define a set of external memory handle types that **may** be used as backing store for an image, add a `VkExternalMemoryImageCreateInfo` structure to the `pNext` chain of the `VkImageCreateInfo` structure. The `VkExternalMemoryImageCreateInfo` structure is defined as:

```
typedef struct VkExternalMemoryImageCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkExternalMemoryHandleTypeFlags   handleTypes;
} VkExternalMemoryImageCreateInfo;
```

or the equivalent

```
typedef VkExternalMemoryImageCreateInfo VkExternalMemoryImageCreateInfoKHR;
```

*Note*



An instance of `VkExternalMemoryImageCreateInfo` must be included in the creation parameters for an image that will be bound to memory that is either exported or imported.

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying one or more external memory handle types.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO`
- `handleTypes` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values
- `handleTypes` **must** not be `0`

If the `pNext` chain includes a `VkExternalMemoryImageCreateInfoNV` structure, then that structure defines a set of external memory handle types that **may** be used as backing store for the image.

The `VkExternalMemoryImageCreateInfoNV` structure is defined as:

```
typedef struct VkExternalMemoryImageCreateInfoNV {  
    VkStructureType           sType;  
    const void*               pNext;  
    VkExternalMemoryHandleTypeFlagsNV handleTypes;  
} VkExternalMemoryImageCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` specifying one or more external memory handle types.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV`
- `handleTypes` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values

To create an image with an `external format`, include an instance of `VkExternalFormatANDROID` in the `pNext` chain of `VkImageCreateInfo`. `VkExternalFormatANDROID` is defined as:

```
typedef struct VkExternalFormatANDROID {
    VkStructureType    sType;
    void*              pNext;
    uint64_t            externalFormat;
} VkExternalFormatANDROID;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `externalFormat` is an implementation-defined identifier for the external format

If `externalFormat` is zero, the effect is as if the `VkExternalFormatANDROID` structure was not present. Otherwise, the `image` will have the specified external format.

## Valid Usage

- `externalFormat` **must** be `0` or a value returned in the `externalFormat` member of `VkAndroidHardwareBufferFormatPropertiesANDROID` by an earlier call to `vkGetAndroidHardwareBufferPropertiesANDROID`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_FORMAT_ANDROID`

If the `pNext` chain of `VkImageCreateInfo` includes a `VkImageSwapchainCreateInfoKHR` structure, then that structure includes a swapchain handle indicating that the image will be bound to memory from that swapchain.

The `VkImageSwapchainCreateInfoKHR` structure is defined as:

```
typedef struct VkImageSwapchainCreateInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
} VkImageSwapchainCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `swapchain` is `VK_NULL_HANDLE` or a handle of a swapchain that the image will be bound to.

## Valid Usage

- If `swapchain` is not `VK_NULL_HANDLE`, the fields of `VkImageCreateInfo` **must** match the implied image creation parameters of the swapchain

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR`
- If `swapchain` is not `VK_NULL_HANDLE`, `swapchain` **must** be a valid `VkSwapchainKHR` handle

If the `pNext` list of `VkImageCreateInfo` includes a `VkImageFormatListCreateInfoKHR` structure, then that structure contains a list of all formats that **can** be used when creating views of this image.

The `VkImageFormatListCreateInfoKHR` structure is defined as:

```
typedef struct VkImageFormatListCreateInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           viewFormatCount;
    const VkFormat*    pViewFormats;
} VkImageFormatListCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is **NULL** or a pointer to an extension-specific structure.
- `viewFormatCount` is the number of entries in the `pViewFormats` array.
- `pViewFormats` is an array which lists of all formats which **can** be used when creating views of this image.

If `viewFormatCount` is zero, `pViewFormats` is ignored and the image is created as if the `VkImageFormatListCreateInfoKHR` structure were not included in the `pNext` list of `VkImageCreateInfo`.

## Valid Usage

- If `viewFormatCount` is not `0`, all of the formats in the `pViewFormats` array **must** be compatible with the format specified in the `format` field of `VkImageCreateInfo`, as described in the compatibility table.
- If `VkImageCreateInfo::flags` does not contain `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, `viewFormatCount` **must** be `0` or `1`.
- If `viewFormatCount` is not `0`, `VkImageCreateInfo::format` **must** be in `pViewFormats`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO_KHR`
- If `viewFormatCount` is not `0`, `pViewFormats` must be a valid pointer to an array of `viewFormatCount` valid `VkFormat` values

If the `pNext` chain of `VkImageCreateInfo` contains `VkImageDrmFormatModifierListCreateInfoEXT`, then the image will be created with one of the [Linux DRM format modifiers](#) listed in the structure. The choice of modifier is implementation-dependent.

The `VkImageDrmFormatModifierListCreateInfoEXT` structure is defined as:

```
typedef struct VkImageDrmFormatModifierListCreateInfoEXT {  
    VkStructureType      sType;  
    const void*          pNext;  
    uint32_t              drmFormatModifierCount;  
    const uint64_t*       pDrmFormatModifiers;  
} VkImageDrmFormatModifierListCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `drmFormatModifierCount` is the length of the `pDrmFormatModifiers` array.
- `pDrmFormatModifiers` is a pointer to an array of *Linux DRM format modifiers*.

## Valid Usage

- Each *modifier* in `pDrmFormatModifiers` must be compatible with the parameters in `VkImageCreateInfo` and its `pNext` chain, as determined by querying `VkPhysicalDeviceImageFormatInfo2` extended with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT`
- `pDrmFormatModifiers` must be a valid pointer to an array of `drmFormatModifierCount` `uint64_t` values
- `drmFormatModifierCount` must be greater than `0`

If the `pNext` chain of `VkImageCreateInfo` contains `VkImageDrmFormatModifierExplicitCreateInfoEXT`, then the image will be created with the [Linux DRM format modifier](#) and memory layout defined by the structure.

The `VkImageDrmFormatModifierExplicitCreateInfoEXT` structure is defined as:

```
typedef struct VkImageDrmFormatModifierExplicitCreateInfoEXT {
    VkStructureType          sType;
    const void*               pNext;
    uint64_t                  drmFormatModifier;
    uint32_t                  drmFormatModifierPlaneCount;
    const VkSubresourceLayout* pPlaneLayouts;
} VkImageDrmFormatModifierExplicitCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `drmFormatModifier` is the *Linux DRM format modifier* with which the image will be created.
- `drmFormatModifierPlaneCount` is the number of *memory planes* in the image (as reported by `VkDrmFormatModifierPropertiesEXT`) as well as the length of the `pPlaneLayouts` array.
- `pPlaneLayouts` is a pointer to an array of `VkSubresourceLayout` structures describing the image's *memory planes*.

The  $i^{\text{th}}$  member of `pPlaneLayouts` describes the layout of the image's  $i^{\text{th}}$  *memory plane* (that is, `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT`). In each element of `pPlaneLayouts`, the implementation **must** ignore `size`. The implementation calculates the size of each plane, which the application **can** query with `vkGetImageSubresourceLayout`.

When creating an image with `VkImageDrmFormatModifierExplicitCreateInfoEXT`, it is the application's responsibility to satisfy all valid usage requirements. However, the implementation **must** validate that the provided `pPlaneLayouts`, when combined with the provided `drmFormatModifier` and other creation parameters in `VkImageCreateInfo` and its `pNext` chain, produce a valid image. (This validation is necessarily implementation-dependent and outside the scope of Vulkan, and therefore not described by valid usage requirements). If this validation fails, then `vkCreateImage` returns `VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT`.

## Valid Usage

- `drmFormatModifier` must be compatible with the parameters in `VkImageCreateInfo` and its `pNext` chain, as determined by querying `VkPhysicalDeviceImageFormatInfo2` extended with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`.
- `drmFormatModifierPlaneCount` **must** be equal to the `VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount` associated with `VkImageCreateInfo::format` and `drmFormatModifier`, as found by querying `VkDrmFormatModifierPropertiesListEXT`.
- For each element of `pPlaneLayouts`, `size` **must** be 0
- For each element of `pPlaneLayouts`, `arrayPitch` **must** be 0 if `VkImageCreateInfo::arrayLayers` is 1.
- For each element of `pPlaneLayouts`, `depthPitch` **must** be 0 if `VkImageCreateInfo::extent::depth` is 1.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT`
- If `drmFormatModifierPlaneCount` is not 0, `pPlaneLayouts` **must** be a valid pointer to an array of `drmFormatModifierPlaneCount` `VkSubresourceLayout` structures

Bits which **can** be set in `VkImageCreateInfo::usage`, specifying intended usage of an image, are:

```
typedef enum VkImageUsageFlagBits {
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_IMAGE_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_IMAGE_USAGE_SAMPLED_BIT = 0x00000004,
    VK_IMAGE_USAGE_STORAGE_BIT = 0x00000008,
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT = 0x00000010,
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000020,
    VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT = 0x00000040,
    VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT = 0x00000080,
    VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00000100,
    VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT = 0x00000200,
    VK_IMAGE_USAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkImageUsageFlagBits;
```

- `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` specifies that the image **can** be used as the source of a transfer command.
- `VK_IMAGE_USAGE_TRANSFER_DST_BIT` specifies that the image **can** be used as the destination of a transfer command.
- `VK_IMAGE_USAGE_SAMPLED_BIT` specifies that the image **can** be used to create a `VkImageView` suitable for occupying a `VkDescriptorSet` slot either of type `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and be sampled by a shader.

- `VK_IMAGE_USAGE_STORAGE_BIT` specifies that the image **can** be used to create a `VkImageView` suitable for occupying a `VkDescriptorSet` slot of type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`.
- `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` specifies that the image **can** be used to create a `VkImageView` suitable for use as a color or resolve attachment in a `VkFramebuffer`.
- `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` specifies that the image **can** be used to create a `VkImageView` suitable for use as a depth/stencil or depth/stencil resolve attachment in a `VkFramebuffer`.
- `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` specifies that the memory bound to this image will have been allocated with the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` (see [Memory Allocation](#) for more detail). This bit **can** be set for any image that **can** be used to create a `VkImageView` suitable for use as a color, resolve, depth/stencil, or input attachment.
- `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` specifies that the image **can** be used to create a `VkImageView` suitable for occupying `VkDescriptorSet` slot of type `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`; be read from a shader as an input attachment; and be used as an input attachment in a framebuffer.
- `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV` specifies that the image **can** be used to create a `VkImageView` suitable for use as a [shading rate image](#).
- `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT` specifies that the image **can** be used to create a `VkImageView` suitable for use as a [fragment density map image](#).

```
typedef VkFlags VkImageUsageFlags;
```

`VkImageUsageFlags` is a bitmask type for setting a mask of zero or more `VkImageUsageFlagBits`.

Bits which **can** be set in `VkImageCreateInfo::flags`, specifying additional parameters of an image, are:

```

typedef enum VkImageCreateFlagBits {
    VK_IMAGE_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_IMAGE_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
    VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT = 0x00000008,
    VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT = 0x00000010,
    VK_IMAGE_CREATE_ALIAS_BIT = 0x00004000,
    VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT = 0x00000040,
    VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT = 0x00000020,
    VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT = 0x00000080,
    VK_IMAGE_CREATE_EXTENDED_USAGE_BIT = 0x00000100,
    VK_IMAGE_CREATE_PROTECTED_BIT = 0x00000800,
    VK_IMAGE_CREATE_DISJOINT_BIT = 0x00000200,
    VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV = 0x00002000,
    VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT = 0x00001000,
    VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT = 0x00004000,
    VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR =
VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT,
    VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT_KHR =
VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT,
    VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT_KHR =
VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT,
    VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR = VK_IMAGE_CREATE_EXTENDED_USAGE_BIT,
    VK_IMAGE_CREATE_DISJOINT_BIT_KHR = VK_IMAGE_CREATE_DISJOINT_BIT,
    VK_IMAGE_CREATE_ALIAS_BIT_KHR = VK_IMAGE_CREATE_ALIAS_BIT,
    VK_IMAGE_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkImageCreateFlagBits;

```

- `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` specifies that the image will be backed using sparse memory binding.
- `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` specifies that the image **can** be partially backed using sparse memory binding. Images created with this flag **must** also be created with the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` flag.
- `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` specifies that the image will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another image (or another portion of the same image). Images created with this flag **must** also be created with the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` flag
- `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` specifies that the image **can** be used to create a `VkImageView` with a different format from the image. For `multi-planar` formats, `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` specifies that a `VkImageView` can be created of a *plane* of the image.
- `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` specifies that the image **can** be used to create a `VkImageView` of type `VK_IMAGE_VIEW_TYPE_CUBE` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`.
- `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` specifies that the image **can** be used to create a `VkImageView` of type `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`.
- `VK_IMAGE_CREATE_PROTECTED_BIT` specifies that the image is a protected image.

- `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` specifies that the image **can** be used with a non-zero value of the `splitInstanceBindRegionCount` member of a `VkBindImageMemoryDeviceGroupInfo` structure passed into `vkBindImageMemory2`. This flag also has the effect of making the image use the standard sparse image block dimensions.
- `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` specifies that the image having a compressed format **can** be used to create a `VkImageView` with an uncompressed format where each texel in the image view corresponds to a compressed texel block of the image.
- `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` specifies that the image **can** be created with usage flags that are not supported for the format the image is created with but are supported for at least one format a `VkImageView` created from the image **can** have.
- `VK_IMAGE_CREATE_DISJOINT_BIT` specifies that an image with a `multi-planar format` **must** have each plane separately bound to memory, rather than having a single memory binding for the whole image; the presence of this bit distinguishes a *disjoint image* from an image without this bit set.
- `VK_IMAGE_CREATE_ALIAS_BIT` specifies that two images created with the same creation parameters and aliased to the same memory **can** interpret the contents of the memory consistently with each other, subject to the rules described in the [Memory Aliasing](#) section. This flag further specifies that each plane of a *disjoint* image **can** share an in-memory non-linear representation with single-plane images, and that a single-plane image **can** share an in-memory non-linear representation with a plane of a multi-planar disjoint image, according to the rules in [Compatible formats of planes of multi-planar formats](#). If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` or `VkExternalMemoryImageCreateInfoNV` structure whose `handleTypes` member is not `0`, it is as if `VK_IMAGE_CREATE_ALIAS_BIT` is set.
- `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` specifies that an image with a depth or depth/stencil format **can** be used with custom sample locations when used as a depth/stencil attachment.
- `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` specifies that the image is a [corner-sampled image](#).
- `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT` specifies that an image **can** be in a subsampled format which **may** be more optimal when written as an attachment by a render pass that has a fragment density map attachment. Accessing a subsampled image has additional considerations:
  - Image data read as an image sampler is undefined if the sampler was not created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` or was not sampled through the use of a combined image sampler with an immutable sampler in `VkDescriptorSetLayoutBinding`.
  - Image data read with an input attachment is undefined if the contents were not written as an attachment in an earlier subpass of the same render pass.
  - Image data read with load operations **may** be resampled to the fragment density of the render pass.
  - Image contents outside of the render area become undefined if the image is stored as a render pass attachment.

See [Sparse Resource Features](#) and [Sparse Physical Device Features](#) for more details.

```
typedef VkFlags VkImageCreateFlags;
```

`VkImageCreateFlags` is a bitmask type for setting a mask of zero or more `VkImageCreateFlagBits`.

Possible values of `VkImageCreateInfo::imageType`, specifying the basic dimensionality of an image, are:

```
typedef enum VkImageType {
    VK_IMAGE_TYPE_1D = 0,
    VK_IMAGE_TYPE_2D = 1,
    VK_IMAGE_TYPE_3D = 2,
    VK_IMAGE_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkImageType;
```

- `VK_IMAGE_TYPE_1D` specifies a one-dimensional image.
- `VK_IMAGE_TYPE_2D` specifies a two-dimensional image.
- `VK_IMAGE_TYPE_3D` specifies a three-dimensional image.

Possible values of `VkImageCreateInfo::tiling`, specifying the tiling arrangement of texel blocks in an image, are:

```
typedef enum VkImageTiling {
    VK_IMAGE_TILING_OPTIMAL = 0,
    VK_IMAGE_TILING_LINEAR = 1,
    VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT = 1000158000,
    VK_IMAGE_TILING_MAX_ENUM = 0x7FFFFFFF
} VkImageTiling;
```

- `VK_IMAGE_TILING_OPTIMAL` specifies optimal tiling (texels are laid out in an implementation-dependent arrangement, for more optimal memory access).
- `VK_IMAGE_TILING_LINEAR` specifies linear tiling (texels are laid out in memory in row-major order, possibly with some padding on each row).
- `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` indicates that the image's tiling is defined by a [Linux DRM format modifier](#). The modifier is specified at image creation with `VkImageDrmFormatModifierListCreateInfoEXT` or `VkImageDrmFormatModifierExplicitCreateInfoEXT`, and can be queried with `vkGetImageDrmFormatModifierPropertiesEXT`.

To query the memory layout of an image subresource, call:

```
void vkGetImageSubresourceLayout(  
    VkDevice device,  
    VkImage image,  
    const VkImageSubresource* pSubresource,  
    VkSubresourceLayout* pLayout);
```

- `device` is the logical device that owns the image.
- `image` is the image whose layout is being queried.
- `pSubresource` is a pointer to a `VkImageSubresource` structure selecting a specific image for the image subresource.
- `pLayout` is a pointer to a `VkSubresourceLayout` structure in which the layout is returned.

If the image is `linear`, then the returned layout is valid for `host access`.

If the image's tiling is `VK_IMAGE_TILING_LINEAR` and its format is a `multi-planar format`, then `vkGetImageSubresourceLayout` describes one *format plane* of the image. If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `vkGetImageSubresourceLayout` describes one *memory plane* of the image. If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and the image is `non-linear`, then the returned layout has an implementation-dependent meaning; the vendor of the image's `DRM format modifier` **may** provide documentation that explains how to interpret the returned layout.

`vkGetImageSubresourceLayout` is invariant for the lifetime of a single image. However, the subresource layout of images in Android hardware buffer external memory is not known until the image has been bound to memory, so applications **must** not call `vkGetImageSubresourceLayout` for such an image before it has been bound.

## Valid Usage

- `image` **must** have been created with `tiling` equal to `VK_IMAGE_TILING_LINEAR` or `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`
- The `aspectMask` member of `pSubresource` **must** only have a single bit set
- The `mipLevel` member of `pSubresource` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- The `arrayLayer` member of `pSubresource` **must** be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- If the `tiling` of the `image` is `VK_IMAGE_TILING_LINEAR` and its `format` is a `multi-planar` format with two planes, the `aspectMask` member of `pSubresource` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`
- If the `tiling` of the `image` is `VK_IMAGE_TILING_LINEAR` and its `format` is a `multi-planar` format with three planes, the `aspectMask` member of `pSubresource` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or `VK_IMAGE_ASPECT_PLANE_2_BIT`
- If `image` was created with the `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` external memory handle type, then `image` **must** be bound to memory.
- If the `tiling` of the `image` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the `aspectMask` member of `pSubresource` **must** be `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` and the index `i` **must** be less than the `drmFormatModifierPlaneCount` associated with the image's `format` and `drmFormatModifier`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `image` **must** be a valid `VkImage` handle
- `pSubresource` **must** be a valid pointer to a valid `VkImageSubresource` structure
- `pLayout` **must** be a valid pointer to a `VkSubresourceLayout` structure
- `image` **must** have been created, allocated, or retrieved from `device`

The `VkImageSubresource` structure is defined as:

```
typedef struct VkImageSubresource {  
    VkImageAspectFlags aspectMask;  
    uint32_t mipLevel;  
    uint32_t arrayLayer;  
} VkImageSubresource;
```

- `aspectMask` is a `VkImageAspectFlags` selecting the image `aspect`.
- `mipLevel` selects the mipmap level.

- `arrayLayer` selects the array layer.

## Valid Usage (Implicit)

- `aspectMask` must be a valid combination of `VkImageAspectFlagBits` values
- `aspectMask` must not be 0

Information about the layout of the image subresource is returned in a `VkSubresourceLayout` structure:

```
typedef struct VkSubresourceLayout {
    VkDeviceSize    offset;
    VkDeviceSize    size;
    VkDeviceSize    rowPitch;
    VkDeviceSize    arrayPitch;
    VkDeviceSize    depthPitch;
} VkSubresourceLayout;
```

- `offset` is the byte offset from the start of the image or the plane where the image subresource begins.
- `size` is the size in bytes of the image subresource. `size` includes any extra memory that is required based on `rowPitch`.
- `rowPitch` describes the number of bytes between each row of texels in an image.
- `arrayPitch` describes the number of bytes between each array layer of an image.
- `depthPitch` describes the number of bytes between each slice of 3D image.

If the image is `linear`, then `rowPitch`, `arrayPitch` and `depthPitch` describe the layout of the image subresource in linear memory. For uncompressed formats, `rowPitch` is the number of bytes between texels with the same x coordinate in adjacent rows (y coordinates differ by one). `arrayPitch` is the number of bytes between texels with the same x and y coordinate in adjacent array layers of the image (array layer values differ by one). `depthPitch` is the number of bytes between texels with the same x and y coordinate in adjacent slices of a 3D image (z coordinates differ by one). Expressed as an addressing formula, the starting byte of a texel in the image subresource has address:

```
// (x,y,z,layer) are in texel coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x*elementSize +
offset
```

For compressed formats, the `rowPitch` is the number of bytes between compressed texel blocks in adjacent rows. `arrayPitch` is the number of bytes between compressed texel blocks in adjacent array layers. `depthPitch` is the number of bytes between compressed texel blocks in adjacent slices of a 3D image.

```
// (x,y,z,layer) are in compressed texel block coordinates  
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x  
*compressedTexelBlockByteSize + offset;
```

The value of `arrayPitch` is undefined for images that were not created as arrays. `depthPitch` is defined only for 3D images.

If the image has a *single-plane* color format and its tiling is `VK_IMAGE_TILING_LINEAR`, then the `aspectMask` member of `VkImageSubresource` **must** be `VK_IMAGE_ASPECT_COLOR_BIT`.

If the image has a depth/stencil format and its tiling is `VK_IMAGE_TILING_LINEAR`, then `aspectMask` **must** be either `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`. On implementations that store depth and stencil aspects separately, querying each of these image subresource layouts will return a different `offset` and `size` representing the region of memory used for that aspect. On implementations that store depth and stencil aspects interleaved, the same `offset` and `size` are returned and represent the interleaved memory allocation.

If the image has a *multi-planar format* and its tiling is `VK_IMAGE_TILING_LINEAR`, then the `aspectMask` member of `VkImageSubresource` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or (for 3-plane formats only) `VK_IMAGE_ASPECT_PLANE_2_BIT`. Querying each of these image subresource layouts will return a different `offset` and `size` representing the region of memory used for that plane. If the image is *disjoint*, then the `offset` is relative to the base address of the plane. If the image is *non-disjoint*, then the `offset` is relative to the base address of the image.

If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the `aspectMask` member of `VkImageSubresource` **must** be one of `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT`, where the maximum allowed plane index `i` is defined by the `drmFormatModifierPlaneCount` associated with the image's `format` and `modifier`. The memory range used by the subresource is described by `offset` and `size`. If the image is *disjoint*, then the `offset` is relative to the base address of the *memory plane*. If the image is *non-disjoint*, then the `offset` is relative to the base address of the image. If the image is *non-linear*, then `rowPitch`, `arrayPitch`, and `depthPitch` have an implementation-dependent meaning.

If an image was created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the image has a [Linux DRM format modifier](#). To query the *modifier*, call:

```
VkResult vkGetImageDrmFormatModifierPropertiesEXT(  
    VkDevice                               device,  
    VkImage                                image,  
    VkImageDrmFormatModifierPropertiesEXT* pProperties);
```

- `device` is the logical device that owns the image.
- `image` is the queried image.
- `pProperties` will return properties of the image's *DRM format modifier*.

## Valid Usage

- **image must have been created with `tiling` equal to `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`.**

## Valid Usage (Implicit)

- **device must be a valid `VkDevice` handle**
- **image must be a valid `VkImage` handle**
- **pProperties must be a valid pointer to a `VkImageDrmFormatModifierPropertiesEXT` structure**
- **image must have been created, allocated, or retrieved from device**

## Return Codes

### Success

- `VK_SUCCESS`

The `VkImageDrmFormatModifierPropertiesEXT` structure is defined as:

```
typedef struct VkImageDrmFormatModifierPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint64_t drmFormatModifier;
} VkImageDrmFormatModifierPropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **drmFormatModifier** returns the image's [Linux DRM format modifier](#).

If the `image` was created with `VkImageDrmFormatModifierListCreateInfoEXT`, then the returned `drmFormatModifier` **must** belong to the list of modifiers provided at time of image creation in `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`. If the `image` was created with `VkImageDrmFormatModifierExplicitCreateInfoEXT`, then the returned `drmFormatModifier` **must** be the modifier provided at time of image creation in `VkImageDrmFormatModifierExplicitCreateInfoEXT::drmFormatModifier`.

## Valid Usage (Implicit)

- **sType must be `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT`**
- **pNext must be `NULL`**

To destroy an image, call:

```
void vkDestroyImage(  
    VkDevice device,  
    VkImage image,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the image.
- `image` is the image to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `image`, either directly or via a `VkImageView`, **must** have completed execution
- If `VkAllocationCallbacks` were provided when `image` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `image` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `image` is not `VK_NULL_HANDLE`, `image` **must** be a valid `VkImage` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `image` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `image` **must** be externally synchronized

### 11.3.1. Image Format Features

Valid usage of a `VkImage` **may** be constrained by the image's format features, defined below. Such constraints are documented in the affected valid usage statement.

- If the image was created with `VK_IMAGE_TILING_LINEAR`, then its set of *format features* is the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageCreateInfo::format`.
- If the image was created with `VK_IMAGE_TILING_OPTIMAL`, but without an `external format`, then its

set of *format features* is the value of `VkFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageCreateInfo::format`.

- If the image was created with an *external format*, then its set of *format features* is the value of `VkAndroidHardwareBufferFormatPropertiesANDROID::formatFeatures` found by calling `vkGetAndroidHardwareBufferPropertiesANDROID` on the Android hardware buffer that was imported to the `VkDeviceMemory` to which the image is bound.
- If the image was created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then:
  - The image's DRM format modifier is the value of `VkImageDrmFormatModifierListCreateInfoEXT::drmFormatModifier` found by calling `vkGetImageDrmFormatModifierPropertiesEXT`.
  - Let `VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties` be the array found by calling `vkGetPhysicalDeviceFormatProperties2` on the same `format` as `VkImageCreateInfo::format`.
  - Let `VkDrmFormatModifierPropertiesEXT prop` be the array element whose `drmFormatModifier` member is the value of the image's DRM format modifier.
  - Then the image set of *format features* is the value of `prop::drmFormatModifierTilingFeatures`.

### 11.3.2. Corner-Sampled Images

A *corner-sampled image* is an image where unnormalized texel coordinates are centered on integer values rather than half-integer values.

A corner-sampled image has a number of differences compared to conventional texture image:

- Texels are centered on integer coordinates. See [Unnormalized Texel Coordinate Operations](#)
- Normalized coordinates are scaled using  $\text{coord} * (\text{dim} - 1)$  rather than  $\text{coord} * \text{dim}$ , where  $\text{dim}$  is the size of one dimension of the image. See [normalized texel coordinate transform](#).
- Partial derivatives are scaled using  $\text{coord} * (\text{dim} - 1)$  rather than  $\text{coord} * \text{dim}$ . See [Scale Factor Operation](#).
- Calculation of the next higher lod size goes according to  $\text{dim} / 2$  rather than  $\text{dim} / 2$ . See [Image Miplevel Sizing](#).
- The minimum level size is 2x2 for 2D images and 2x2x2 for 3D images. See [Image Miplevel Sizing](#).

Corner-sampling is only supported for 2D and 3D images. When sampling a corner-sampled image, the sampler addressing mode **must** be `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`. Corner-sampled images are not supported as cubemaps or depth/stencil images.

### 11.3.3. Image Miplevel Sizing

A *complete mipmap chain* is the full set of miplevels, from the largest miplevel provided, down to the *minimum miplevel size*.

## Conventional Images

For conventional images, the dimensions of each successive miplevel,  $n+1$ , are:

$$\text{width}_{n+1} = \max(\text{width}_n/2, 1)$$

$$\text{height}_{n+1} = \max(\text{height}_n/2, 1)$$

$$\text{depth}_{n+1} = \max(\text{depth}_n/2, 1)$$

where  $\text{width}_n$ ,  $\text{height}_n$ , and  $\text{depth}_n$  are the dimensions of the next larger miplevel,  $n$ .

The minimum miplevel size is:

- 1 for one-dimensional images,
- 1x1 for two-dimensional images, and
- 1x1x1 for three-dimensional images.

The number of levels in a complete mipmap chain is:

$$\log_2(\max(\text{width}_0, \text{height}_0, \text{depth}_0)) + 1$$

where  $\text{width}_0$ ,  $\text{height}_0$ , and  $\text{depth}_0$  are the dimensions of the largest (most detailed) miplevel,  $0$ .

## Corner-Sampled Images

For corner-sampled images, the dimensions of each successive miplevel,  $n+1$ , are:

$$\text{width}_{n+1} = \max(\text{width}_n/2, 2)$$

$$\text{height}_{n+1} = \max(\text{height}_n/2, 2)$$

$$\text{depth}_{n+1} = \max(\text{depth}_n/2, 2)$$

where  $\text{width}_n$ ,  $\text{height}_n$ , and  $\text{depth}_n$  are the dimensions of the next larger miplevel,  $n$ .

The minimum miplevel size is:

- 2x2 for two-dimensional images, and
- 2x2x2 for three-dimensional images.

The number of levels in a complete mipmap chain is:

$$\log_2(\max(\text{width}_0, \text{height}_0, \text{depth}_0))$$

where  $\text{width}_0$ ,  $\text{height}_0$ , and  $\text{depth}_0$  are the dimensions of the largest (most detailed) miplevel,  $0$ .

## 11.4. Image Layouts

Images are stored in implementation-dependent opaque layouts in memory. Each layout has limitations on what kinds of operations are supported for image subresources using the layout. At any given time, the data representing an image subresource in memory exists in a particular layout which is determined by the most recent layout transition that was performed on that image subresource. Applications have control over which layout each image subresource uses, and **can** transition an image subresource from one layout to another. Transitions **can** happen with an image memory barrier, included as part of a `vkCmdPipelineBarrier` or a `vkCmdWaitEvents` command buffer command (see [Image Memory Barriers](#)), or as part of a subpass dependency within a render pass (see [VkSubpassDependency](#)). The image layout is per-image subresource, and separate image subresources of the same image **can** be in different layouts at the same time with one exception - depth and stencil aspects of a given image subresource **must** always be in the same layout.

*Note*

Each layout **may** offer optimal performance for a specific usage of image memory. For example, an image with a layout of `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` **may** provide optimal performance for use as a color attachment, but be unsupported for use in transfer commands. Applications **can** transition an image subresource from one layout to another in order to achieve optimal performance when the image subresource is used for multiple kinds of operations. After initialization, applications need not use any layout other than the general layout, though this **may** produce suboptimal performance on some implementations.



Upon creation, all image subresources of an image are initially in the same layout, where that layout is selected by the `VkImageCreateInfo::initialLayout` member. The `initialLayout` **must** be either `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`. If it is `VK_IMAGE_LAYOUT_PREINITIALIZED`, then the image data **can** be preinitialized by the host while using this layout, and the transition away from this layout will preserve that data. If it is `VK_IMAGE_LAYOUT_UNDEFINED`, then the contents of the data are considered to be undefined, and the transition away from this layout is not guaranteed to preserve that data. For either of these initial layouts, any image subresources **must** be transitioned to another layout before they are accessed by the device.

Host access to image memory is only well-defined for [\[glossary-linear-resource\]](#) images and for image subresources of those images which are currently in either the `VK_IMAGE_LAYOUT_PREINITIALIZED` or `VK_IMAGE_LAYOUT_GENERAL` layout. Calling `vkGetImageSubresourceLayout` for a linear image returns a subresource layout mapping that is valid for either of those image layouts.

The set of image layouts consists of:

```

typedef enum VkImageLayout {
    VK_IMAGE_LAYOUT_UNDEFINED = 0,
    VK_IMAGE_LAYOUT_GENERAL = 1,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL = 2,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL = 3,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL = 4,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL = 5,
    VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL = 6,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL = 7,
    VK_IMAGE_LAYOUT_PREINITIALIZED = 8,
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL = 1000117000,
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL = 1000117001,
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR = 1000001002,
    VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR = 1000111000,
    VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV = 1000164003,
    VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT = 1000218000,
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR = 1000241000,
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR = 1000241001,
    VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR = 1000241002,
    VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR = 1000241003,
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL_KHR =
VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL,
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL_KHR =
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
    VK_IMAGE_LAYOUT_MAX_ENUM = 0x7FFFFFFF
} VkImageLayout;

```

The type(s) of device access supported by each layout are:

- **VK\_IMAGE\_LAYOUT\_UNDEFINED** does not support device access. This layout **must** only be used as the `initialLayout` member of `VkImageCreateInfo` or `VkAttachmentDescription`, or as the `oldLayout` in an image transition. When transitioning out of this layout, the contents of the memory are not guaranteed to be preserved.
- **VK\_IMAGE\_LAYOUT\_PREINITIALIZED** does not support device access. This layout **must** only be used as the `initialLayout` member of `VkImageCreateInfo` or `VkAttachmentDescription`, or as the `oldLayout` in an image transition. When transitioning out of this layout, the contents of the memory are preserved. This layout is intended to be used as the initial layout for an image whose contents are written by the host, and hence the data **can** be written to memory immediately, without first executing a layout transition. Currently, **VK\_IMAGE\_LAYOUT\_PREINITIALIZED** is only useful with `linear` images because there is not a standard layout defined for **VK\_IMAGE\_TILING\_OPTIMAL** images.
- **VK\_IMAGE\_LAYOUT\_GENERAL** supports all types of device access.
- **VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL** **must** only be used as a color or resolve attachment in a `VkFramebuffer`. This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` usage bit enabled.
- **VK\_IMAGE\_LAYOUT\_DEPTH\_STENCIL\_ATTACHMENT\_OPTIMAL** specifies a layout for both the depth and stencil aspects of a depth/stencil format image allowing read and write access as a depth/stencil

attachment. It is equivalent to `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` and `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`.

- `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` specifies a layout for both the depth and stencil aspects of a depth/stencil format image allowing read only access as a depth/stencil attachment or in shaders. It is equivalent to `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR` and `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`.
- `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` specifies a layout for depth/stencil format images allowing read and write access to the stencil aspect as a stencil attachment, and read only access to the depth aspect as a depth attachment or in shaders. It is equivalent to `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR` and `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`.
- `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` specifies a layout for depth/stencil format images allowing read and write access to the depth aspect as a depth attachment, and read only access to the stencil aspect as a stencil attachment or in shaders. It is equivalent to `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` and `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`.
- `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR` specifies a layout for the depth aspect of a depth/stencil format image allowing read and write access as a depth attachment.
- `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR` specifies a layout for the depth aspect of a depth/stencil format image allowing read-only access as a depth attachment or in shaders.
- `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR` specifies a layout for the stencil aspect of a depth/stencil format image allowing read and write access as a stencil attachment.
- `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` specifies a layout for the stencil aspect of a depth/stencil format image allowing read-only access as a stencil attachment or in shaders.
- `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` **must** only be used as a read-only image in a shader (which **can** be read as a sampled image, combined image/sampler and/or input attachment). This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` usage bit enabled.
- `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` **must** only be used as a source image of a transfer command (see the definition of `VK_PIPELINE_STAGE_TRANSFER_BIT`). This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage bit enabled.
- `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` **must** only be used as a destination image of a transfer command. This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage bit enabled.
- `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` **must** only be used for presenting a presentable image for display. A swapchain's image **must** be transitioned to this layout before calling `vkQueuePresentKHR`, and **must** be transitioned away from this layout after calling `vkAcquireNextImageKHR`.
- `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` is valid only for shared presentable images, and **must** be used for any usage the image supports.
- `VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV` **must** only be used as a read-only [shading-rate-image](#). This layout is valid only for image subresources of images created with the

`VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV` usage bit enabled.

- `VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT` **must** only be used as a fragment density map attachment in a `VkRenderPass`. This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT` usage bit enabled.

The layout of each image subresource is not a state of the image subresource itself, but is rather a property of how the data in memory is organized, and thus for each mechanism of accessing an image in the API the application **must** specify a parameter or structure member that indicates which image layout the image subresource(s) are considered to be in when the image will be accessed. For transfer commands, this is a parameter to the command (see [Clear Commands](#) and [Copy Commands](#)). For use as a framebuffer attachment, this is a member in the substructures of the `VkRenderPassCreateInfo` (see [Render Pass](#)). For use in a descriptor set, this is a member in the `VkDescriptorImageInfo` structure (see [Descriptor Set Updates](#)).

#### 11.4.1. Image Layout Matching Rules

At the time that any command buffer command accessing an image executes on any queue, the layouts of the image subresources that are accessed **must** all match exactly the layout specified via the API controlling those accesses , except in case of accesses to an image with a depth/stencil format performed through descriptors referring to only a single aspect of the image, where the following relaxed matching rules apply:

- Descriptors referring just to the depth aspect of a depth/stencil image only need to match in the image layout of the depth aspect, thus `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` and `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` are considered to match.
- Descriptors referring just to the stencil aspect of a depth/stencil image only need to match in the image layout of the stencil aspect, thus `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` and `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` are considered to match .

When performing a layout transition on an image subresource, the old layout value **must** either equal the current layout of the image subresource (at the time the transition executes), or else be `VK_IMAGE_LAYOUT_UNDEFINED` (implying that the contents of the image subresource need not be preserved). The new layout used in a transition **must** not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

The image layout of each image subresource of a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` is dependent on the last sample locations used to render to the image subresource as a depth/stencil attachment, thus applications **must** provide the same sample locations that were last used to render to the given image subresource whenever a layout transition of the image subresource happens, otherwise the contents of the depth aspect of the image subresource become undefined.

In addition, depth reads from a depth/stencil attachment referring to an image subresource range of a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` using different sample locations than what have been last used to perform depth writes to the image subresources of the same image subresource range return undefined values.

Similarly, depth writes to a depth/stencil attachment referring to an image subresource range of a

depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` using different sample locations than what have been last used to perform depth writes to the image subresources of the same image subresource range make the contents of the depth aspect of those image subresources undefined.

## 11.5. Image Views

Image objects are not directly accessed by pipeline shaders for reading or writing image data. Instead, *image views* representing contiguous ranges of the image subresources and containing additional metadata are used for that purpose. Views **must** be created on images of compatible types, and **must** represent a valid subset of image subresources.

Image views are represented by `VkImageView` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImageView)
```

The types of image views that **can** be created are:

```
typedef enum VkImageViewType {
    VK_IMAGE_VIEW_TYPE_1D = 0,
    VK_IMAGE_VIEW_TYPE_2D = 1,
    VK_IMAGE_VIEW_TYPE_3D = 2,
    VK_IMAGE_VIEW_TYPE_CUBE = 3,
    VK_IMAGE_VIEW_TYPE_1D_ARRAY = 4,
    VK_IMAGE_VIEW_TYPE_2D_ARRAY = 5,
    VK_IMAGE_VIEW_TYPE_CUBE_ARRAY = 6,
    VK_IMAGE_VIEW_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkImageViewType;
```

The exact image view type is partially implicit, based on the image's type and sample count, as well as the view creation parameters as described in the [image view compatibility table](#) for `vkCreateImageView`. This table also shows which SPIR-V `OpTypeImage` `Dim` and `Arrayed` parameters correspond to each image view type.

To create an image view, call:

```
VkResult vkCreateImageView(
    VkDevice                                     device,
    const VkImageViewCreateInfo*                 pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkImageView*                                pView);
```

- `device` is the logical device that creates the image view.
- `pCreateInfo` is a pointer to a `VkImageViewCreateInfo` structure containing parameters to be used to create the image view.

- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pView` is a pointer to a `VkImageView` handle in which the resulting image view object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkImageViewCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pView` **must** be a valid pointer to a `VkImageView` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkImageViewCreateInfo` structure is defined as:

```
typedef struct VkImageViewCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkImageViewCreateFlags    flags;
    VkImage                  image;
    VkImageViewType           viewType;
    VkFormat                 format;
    VkComponentMapping        components;
    VkImageSubresourceRange   subresourceRange;
} VkImageViewCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkImageViewCreateFlagBits` describing additional parameters of the image view.
- `image` is a `VkImage` on which the view will be created.
- `viewType` is a `VkImageViewType` value specifying the type of the image view.
- `format` is a `VkFormat` describing the format and type used to interpret texel blocks in the image.
- `components` is a `VkComponentMapping` specifies a remapping of color components (or of depth

or stencil components after they have been converted into color components).

- `subresourceRange` is a `VkImageSubresourceRange` selecting the set of mipmap levels and array layers to be accessible to the view.

Some of the `image` creation parameters are inherited by the view. In particular, image view creation inherits the implicit parameter `usage` specifying the allowed usages of the image view that, by default, takes the value of the corresponding `usage` parameter specified in `VkImageCreateInfo` at image creation time. If the image was has a depth-stencil format and was created with an instance of `VkImageStencilUsageCreateInfoEXT` in the `pNext` chain of `VkImageCreateInfo`, the usage is calculated based on the `subresource.aspectMask` provided:

- If `aspectMask` includes only `VK_IMAGE_ASPECT_STENCIL_BIT`, the implicit `usage` is equal to `VkImageStencilUsageCreateInfoEXT::stencilUsage`.
- If `aspectMask` includes only `VK_IMAGE_ASPECT_DEPTH_BIT`, the implicit `usage` is equal to `VkImageCreateInfo::usage`.
- If both aspects are included in `aspectMask`, the implicit `usage` is equal to the intersection of `VkImageCreateInfo::usage` and `VkImageStencilUsageCreateInfoEXT::stencilUsage`. The implicit `usage` can be overridden by including an instance of `VkImageViewUsageCreateInfo` structure in the `pNext` chain.

If `image` was created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` flag, and if the `format` of the image is not `multi-planar`, `format` can be different from the image's format, but if `image` was created without the `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` flag and they are not equal they **must** be *compatible*. Image format compatibility is defined in the [Format Compatibility Classes](#) section. Views of compatible formats will have the same mapping between texel coordinates and memory locations irrespective of the `format`, with only the interpretation of the bit pattern changing.

**Note**

Values intended to be used with one view format **may** not be exactly preserved when written or read through a different format. For example, an integer value that happens to have the bit pattern of a floating point denorm or NaN **may** be flushed or canonicalized when written or read through a view with a floating point format. Similarly, a value written through a signed normalized format that has a bit pattern exactly equal to  $-2^b$  **may** be changed to  $-2^b + 1$  as described in [Conversion from Normalized Fixed-Point to Floating-Point](#).



If `image` was created with the `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` flag, `format` **must** be *compatible* with the image's format as described above, or **must** be an uncompressed format in which case it **must** be *size-compatible* with the image's format, as defined for [copying data between images](#). In this case the resulting image view's texel dimensions equal the dimensions of the selected mip level divided by the compressed texel block size and rounded up.

If the image view is to be used with a sampler which supports `sampler Y'CBCR` conversion, an identically defined object of type `VkSamplerYcbcrConversion` to that used to create the sampler **must** be passed to `vkCreateImageView` in a `VkSamplerYcbcrConversionInfo` added to the `pNext` chain of `VkImageViewCreateInfo`. Conversely, if a `VkSamplerYcbcrConversion` object is passed to `vkCreateImageView`, an identically defined `VkSamplerYcbcrConversion` object **must** be used when

sampling the image.

If the image has a multi-planar format and `subresourceRange.aspectMask` is `VK_IMAGE_ASPECT_COLOR_BIT`, `format` must be identical to the image `format`, and the sampler to be used with the image view must enable sampler Y'CbCr conversion.

If `image` was created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` and the image has a multi-planar format, and if `subresourceRange.aspectMask` is `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`, `format` must be compatible with the corresponding plane of the image, and the sampler to be used with the image view must not enable sampler Y'CbCr conversion. The `width` and `height` of the single-plane image view must be derived from the multi-planar image's dimensions in the manner listed for `plane compatibility` for the plane.

Any view of an image plane will have the same mapping between texel coordinates and memory locations as used by the channels of the color aspect, subject to the formulae relating texel coordinates to lower-resolution planes as described in [Chroma Reconstruction](#). That is, if an R or B plane has a reduced resolution relative to the G plane of the multi-planar image, the image view operates using the  $(u_{\text{plane}}, v_{\text{plane}})$  unnormalized coordinates of the reduced-resolution plane, and these coordinates access the same memory locations as the  $(u_{\text{color}}, v_{\text{color}})$  unnormalized coordinates of the color aspect for which chroma reconstruction operations operate on the same  $(u_{\text{plane}}, v_{\text{plane}})$  or  $(i_{\text{plane}}, j_{\text{plane}})$  coordinates.

*Table 15. Image and image view parameter compatibility requirements*

Dim, Arrayed, MS	Image parameters	View parameters
	<code>imageType</code> = <code>ci.imageType</code> <code>width</code> = <code>ci.extent.width</code> <code>height</code> = <code>ci.extent.height</code> <code>depth</code> = <code>ci.extent.depth</code> <code>arrayLayers</code> = <code>ci.arrayLayers</code> <code>samples</code> = <code>ci.samples</code> <code>flags</code> = <code>ci.flags</code> where <code>ci</code> is the <code>VkImageCreateInfo</code> used to create image.	<code>baseArrayLayer</code> , <code>layerCount</code> , and <code>levelCount</code> are members of the <code>subresourceRange</code> member.
<b>1D, 0, 0</b>	<code>imageType</code> = <code>VK_IMAGE_TYPE_1D</code> <code>width</code> $\geq 1$ <code>height</code> = 1 <code>depth</code> = 1 <code>arrayLayers</code> $\geq 1$ <code>samples</code> = 1	<code>viewType</code> = <code>VK_IMAGE_VIEW_TYPE_1D</code> <code>baseArrayLayer</code> $\geq 0$ <code>layerCount</code> = 1
<b>1D, 1, 0</b>	<code>imageType</code> = <code>VK_IMAGE_TYPE_1D</code> <code>width</code> $\geq 1$ <code>height</code> = 1 <code>depth</code> = 1 <code>arrayLayers</code> $\geq 1$ <code>samples</code> = 1	<code>viewType</code> = <code>VK_IMAGE_VIEW_TYPE_1D_ARRAY</code> <code>baseArrayLayer</code> $\geq 0$ <code>layerCount</code> $\geq 1$

Dim, Arrayed, MS	Image parameters	View parameters
<b>2D, 0, 0</b>	<code>imageType = VK_IMAGE_TYPE_2D</code> <code>width ≥ 1</code> <code>height ≥ 1</code> <code>depth = 1</code> <code>arrayLayers ≥ 1</code> <code>samples = 1</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_2D</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount = 1</code>
<b>2D, 1, 0</b>	<code>imageType = VK_IMAGE_TYPE_2D</code> <code>width ≥ 1</code> <code>height ≥ 1</code> <code>depth = 1</code> <code>arrayLayers ≥ 1</code> <code>samples = 1</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount ≥ 1</code>
<b>2D, 0, 1</b>	<code>imageType = VK_IMAGE_TYPE_2D</code> <code>width ≥ 1</code> <code>height ≥ 1</code> <code>depth = 1</code> <code>arrayLayers ≥ 1</code> <code>samples &gt; 1</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_2D</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount = 1</code>
<b>2D, 1, 1</b>	<code>imageType = VK_IMAGE_TYPE_2D</code> <code>width ≥ 1</code> <code>height ≥ 1</code> <code>depth = 1</code> <code>arrayLayers ≥ 1</code> <code>samples &gt; 1</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount ≥ 1</code>
<b>CUBE, 0, 0</b>	<code>imageType = VK_IMAGE_TYPE_2D</code> <code>width ≥ 1</code> <code>height = width</code> <code>depth = 1</code> <code>arrayLayers ≥ 6</code> <code>samples = 1</code> <code>flags includes VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_CUBE</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount = 6</code>
<b>CUBE, 1, 0</b>	<code>imageType = VK_IMAGE_TYPE_2D</code> <code>width ≥ 1</code> <code>height = width</code> <code>depth = 1</code> <code>N ≥ 1</code> <code>arrayLayers ≥ 6 × N</code> <code>samples = 1</code> <code>flags includes VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_CUBE_ARRAY</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount = 6 × N, N ≥ 1</code>

Dim, Arrayed, MS	Image parameters	View parameters
<b>3D, 0, 0</b>	<code>imageType = VK_IMAGE_TYPE_3D</code> <code>width ≥ 1</code> <code>height ≥ 1</code> <code>depth ≥ 1</code> <code>arrayLayers = 1</code> <code>samples = 1</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_3D</code> <code>baseArrayLayer = 0</code> <code>layerCount = 1</code>
<b>3D, 0, 0</b>	<code>imageType = VK_IMAGE_TYPE_3D</code> <code>width ≥ 1</code> <code>height ≥ 1</code> <code>depth ≥ 1</code> <code>arrayLayers = 1</code> <code>samples = 1</code> <code>flags includes VK_IMAGE_CREATE_2D_ARRAY_COMPATIBILITY_BIT</code> <code>flags does not include VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_2D</code> <code>levelCount = 1</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount = 1</code>
<b>3D, 0, 0</b>	<code>imageType = VK_IMAGE_TYPE_3D</code> <code>width ≥ 1</code> <code>height ≥ 1</code> <code>depth ≥ 1</code> <code>arrayLayers = 1</code> <code>samples = 1</code> <code>flags includes VK_IMAGE_CREATE_2D_ARRAY_COMPATIBILITY_BIT</code> <code>flags does not include VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT</code>	<code>viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY</code> <code>levelCount = 1</code> <code>baseArrayLayer ≥ 0</code> <code>layerCount ≥ 1</code>

## Valid Usage

- If `image` was not created with `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` then `viewType` **must** not be `VK_IMAGE_VIEW_TYPE_CUBE` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`
- If the `image` `cubemap arrays` feature is not enabled, `viewType` **must** not be `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`
- If `image` was created with `VK_IMAGE_TYPE_3D` but without `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set then `viewType` **must** not be `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`
- `image` **must** have been created with a `usage` value containing at least one of `VK_IMAGE_USAGE_SAMPLED_BIT`, `VK_IMAGE_USAGE_STORAGE_BIT`, `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`, or `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`
- The `format features` of the resultant image view **must** contain at least one bit.
- If `usage` contains `VK_IMAGE_USAGE_SAMPLED_BIT`, then the `format features` of the resultant image view **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`.
- If `usage` contains `VK_IMAGE_USAGE_STORAGE_BIT`, then the image view's `format features` **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`.
- If `usage` contains `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, then the image view's `format features` **must** contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.
- If `usage` contains `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, then the image view's `format features` **must** contain `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.
- If `usage` contains `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, then the image view's `format features` **must** contain at least one of `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` or `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.
- `subresourceRange.baseMipLevel` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- If `subresourceRange.levelCount` is not `VK_REMAINING_MIP_LEVELS`, `subresourceRange.baseMipLevel + subresourceRange.levelCount` **must** be less than or equal to the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- If `image` was created with `usage` containing `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `subresourceRange.levelCount` **must** be 1
- If `image` is not a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set, or `viewType` is not `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`, `subresourceRange::baseArrayLayer` **must** be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- If `subresourceRange::layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, `image` is not a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set, or `viewType` is not `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`, `subresourceRange::layerCount` **must** be non-zero and `subresourceRange::baseArrayLayer + subresourceRange::layerCount` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `image` was

created

- If `image` is a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set, and `viewType` is `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`, `subresourceRange::baseArrayLayer` **must** be less than the depth computed from `baseMipLevel` and `extent.depth` specified in `VkImageCreateInfo` when `image` was created, according to the formula defined in [Image Miplevel Sizing](#).
- If `subresourceRange::layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, `image` is a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set, and `viewType` is `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`, `subresourceRange::layerCount` **must** be non-zero and `subresourceRange::baseArrayLayer + subresourceRange::layerCount` **must** be less than or equal to the depth computed from `baseMipLevel` and `extent.depth` specified in `VkImageCreateInfo` when `image` was created, according to the formula defined in [Image Miplevel Sizing](#).
- If `image` was created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` flag, `format` **must** be compatible with the `format` used to create `image`, as defined in [Format Compatibility Classes](#)
- If `image` was created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` flag, but without the `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` flag, and if the `format` of the `image` is not a `multi-planar` format, `format` **must** be compatible with the `format` used to create `image`, as defined in [Format Compatibility Classes](#)
- If `image` was created with the `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` flag, `format` **must** be compatible with, or **must** be an uncompressed format that is size-compatible with, the `format` used to create `image`.
- If `image` was created with the `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT` flag, the `levelCount` and `layerCount` members of `subresourceRange` **must** both be 1.
- If a `VkImageFormatListCreateInfoKHR` structure was included in the `pNext` chain of the `VkImageCreateInfo` structure used when creating `image` and the `viewFormatCount` field of `VkImageFormatListCreateInfoKHR` is not zero then `format` **must** be one of the formats in `VkImageFormatListCreateInfoKHR::pViewFormats`.
- If `image` was created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` flag, if the `format` of the `image` is a `multi-planar` format, and if `subresourceRange.aspectMask` is one of `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`, then `format` **must** be compatible with the `VkFormat` for the plane of the `image` format indicated by `subresourceRange.aspectMask`, as defined in [Compatible formats of planes of multi-planar formats](#)
- If `image` was not created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` flag, or if the `format` of the `image` is a `multi-planar` format and if `subresourceRange.aspectMask` is `VK_IMAGE_ASPECT_COLOR_BIT`, `format` **must** be identical to the `format` used to create `image`
- If the `pNext` chain contains an instance of `VkSamplerYcbcrConversionInfo` with a `conversion` value other than `VK_NULL_HANDLE`, all members of `components` **must** have the value `VK_COMPONENT_SWIZZLE_IDENTITY`.
- If `image` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- `subresourceRange` and `viewType` **must** be compatible with the image, as described in the compatibility table
- If `image` has an `external format`, `format` **must** be `VK_FORMAT_UNDEFINED`.
- If `image` has an `external format`, the `pNext` chain **must** contain an instance of `VkSamplerYcbcrConversionInfo` with a `conversion` object created with the same external format as `image`.
- If `image` has an `external format`, all members of `components` **must** be `VK_COMPONENT_SWIZZLE_IDENTITY`.
- If `image` was created with `usage` containing `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`, `viewType` **must** be `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`
- If `image` was created with `usage` containing `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`, `format` **must** be `VK_FORMAT_R8_UINT`
- If `dynamic fragment density map` feature is not enabled, `flags` **must** not contain `VK_IMAGE_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT`
- If `dynamic fragment density map` feature is not enabled and `image` was created with `usage` containing `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `flags` **must** not contain any of `VK_IMAGE_CREATE_PROTECTED_BIT`, `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`
- If the `pNext` chain includes an instance of `VkImageViewUsageCreateInfo`, and `image` was not created with an instance of `VkImageStencilUsageCreateInfoEXT` in the `pNext` chain of `VkImageCreateInfo`, its `usage` member **must** not include any bits that were not set in the `usage` member of the `VkImageCreateInfo` structure used to create `image`
- If the `pNext` chain includes an instance of `VkImageViewUsageCreateInfo`, `image` was created with an instance of `VkImageStencilUsageCreateInfoEXT` in the `pNext` chain of `VkImageCreateInfo`, and `subResourceRange.aspectMask` includes `VK_IMAGE_ASPECT_STENCIL_BIT`, the `usage` member of the `VkImageViewUsageCreateInfo` instance **must** not include any bits that were not set in the `usage` member of the `VkImageStencilUsageCreateInfoEXT` structure used to create `image`
- If the `pNext` chain includes an instance of `VkImageViewUsageCreateInfo`, `image` was created with an instance of `VkImageStencilUsageCreateInfoEXT` in the `pNext` chain of `VkImageCreateInfo`, and `subResourceRange.aspectMask` includes bits other than `VK_IMAGE_ASPECT_STENCIL_BIT`, the `usage` member of the `VkImageViewUsageCreateInfo` instance **must** not include any bits that were not set in the `usage` member of the `VkImageCreateInfo` structure used to create `image`

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO`
- Each **pNext** member of any structure (including this one) in the **pNext** chain must be either `NULL` or a pointer to a valid instance of `VkImageViewASTCDecodeModeEXT`, `VkImageViewUsageCreateInfo`, or `VkSamplerYcbcrConversionInfo`
- Each **sType** member in the **pNext** chain must be unique
- **flags** must be a valid combination of `VkImageViewCreateFlagBits` values
- **image** must be a valid `VkImage` handle
- **viewType** must be a valid `VkImageViewType` value
- **format** must be a valid `VkFormat` value
- **components** must be a valid `VkComponentMapping` structure
- **subresourceRange** must be a valid `VkImageSubresourceRange` structure

Bits which can be set in `VkImageViewCreateInfo::flags`, specifying additional parameters of an image, are:

```
typedef enum VkImageViewCreateFlagBits {
    VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT = 0x00000001,
    VK_IMAGE_VIEW_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkImageViewCreateFlagBits;
```

- `VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT` prohibits the implementation from accessing the fragment density map by the host during `vkCmdBeginRenderPass` as the contents are expected to change after recording

```
typedef VkFlags VkImageViewCreateFlags;
```

`VkImageViewCreateFlags` is a bitmask type for setting a mask of zero or more `VkImageViewCreateFlagBits`.

The set of usages for the created image view can be restricted compared to the parent image's **usage** flags by chaining a `VkImageViewUsageCreateInfo` structure through the **pNext** member to `VkImageViewCreateInfo`.

The `VkImageViewUsageCreateInfo` structure is defined as:

```
typedef struct VkImageViewUsageCreateInfo {
    VkStructureType      sType;
    const void*          pNext;
    VkImageUsageFlags    usage;
} VkImageViewUsageCreateInfo;
```

or the equivalent

```
typedef VkImageViewUsageCreateInfo VkImageViewUsageCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `usage` is a bitmask describing the allowed usages of the image view. See [VkImageUsageFlagBits](#) for a description of the supported bits.

When this structure is chained to [VkImageViewCreateInfo](#) the `usage` field overrides the implicit `usage` parameter inherited from image creation time and its value is used instead for the purposes of determining the valid usage conditions of [VkImageViewCreateInfo](#).

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO`
- `usage` **must** be a valid combination of [VkImageUsageFlagBits](#) values
- `usage` **must** not be `0`

The [VkImageSubresourceRange](#) structure is defined as:

```
typedef struct VkImageSubresourceRange {  
    VkImageAspectFlags    aspectMask;  
    uint32_t              baseMipLevel;  
    uint32_t              levelCount;  
    uint32_t              baseArrayLayer;  
    uint32_t              layerCount;  
} VkImageSubresourceRange;
```

- `aspectMask` is a bitmask of [VkImageAspectFlagBits](#) specifying which aspect(s) of the image are included in the view.
- `baseMipLevel` is the first mipmap level accessible to the view.
- `levelCount` is the number of mipmap levels (starting from `baseMipLevel`) accessible to the view.
- `baseArrayLayer` is the first array layer accessible to the view.
- `layerCount` is the number of array layers (starting from `baseArrayLayer`) accessible to the view.

The number of mipmap levels and array layers **must** be a subset of the image subresources in the image. If an application wants to use all mip levels or layers in an image after the `baseMipLevel` or `baseArrayLayer`, it **can** set `levelCount` and `layerCount` to the special values `VK_REMAINING_MIP_LEVELS` and `VK_REMAINING_ARRAY_LAYERS` without knowing the exact number of mip levels or layers.

For cube and cube array image views, the layers of the image view starting at `baseArrayLayer` correspond to faces in the order `+X, -X, +Y, -Y, +Z, -Z`. For cube arrays, each set of six sequential

layers is a single cube, so the number of cube maps in a cube map array view is `layerCount` / 6, and image array layer (`baseArrayLayer` + i) is face index (i mod 6) of cube  $i / 6$ . If the number of layers in the view, whether set explicitly in `layerCount` or implied by `VK_REMAINING_ARRAY_LAYERS`, is not a multiple of 6, the last cube map in the array **must** not be accessed.

`aspectMask` **must** be only `VK_IMAGE_ASPECT_COLOR_BIT`, `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` if `format` is a color, depth-only or stencil-only format, respectively, except if `format` is a multi-planar format. If using a depth/stencil format with both depth and stencil components, `aspectMask` **must** include at least one of `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT`, and **can** include both.

When the `VkImageSubresourceRange` structure is used to select a subset of the slices of a 3D image's mip level in order to create a 2D or 2D array image view of a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`, `baseArrayLayer` and `layerCount` specify the first slice index and the number of slices to include in the created image view. Such an image view **can** be used as a framebuffer attachment that refers only to the specified range of slices of the selected mip level. However, any layout transitions performed on such an attachment view during a render pass instance still apply to the entire subresource referenced which includes all the slices of the selected mip level.

When using an image view of a depth/stencil image to populate a descriptor set (e.g. for sampling in the shader, or for use as an input attachment), the `aspectMask` **must** only include one bit and selects whether the image view is used for depth reads (i.e. using a floating-point sampler or input attachment in the shader) or stencil reads (i.e. using an unsigned integer sampler or input attachment in the shader). When an image view of a depth/stencil image is used as a depth/stencil framebuffer attachment, the `aspectMask` is ignored and both depth and stencil image subresources are used.

The `VkComponentMapping` `components` member describes a remapping from components of the image to components of the vector returned by shader image instructions. This remapping **must** be identity for storage image descriptors, input attachment descriptors, framebuffer attachments, and any `VkImageView` used with a combined image sampler that enables `sampler Y'CBCR` conversion.

When creating a `VkImageView`, if `sampler Y'CBCR` conversion is enabled in the sampler, the `aspectMask` of a `subresourceRange` used by the `VkImageView` **must** be `VK_IMAGE_ASPECT_COLOR_BIT`.

When creating a `VkImageView`, if `sampler Y'CBCR` conversion is not enabled in the sampler and the image `format` is multi-planar, the image **must** have been created with `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, and the `aspectMask` of the `VkImageView`'s `subresourceRange` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or `VK_IMAGE_ASPECT_PLANE_2_BIT`.

## Valid Usage

- If `levelCount` is not `VK_REMAINING_MIP_LEVELS`, it **must** be greater than `0`
- If `layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, it **must** be greater than `0`
- If `aspectMask` includes `VK_IMAGE_ASPECT_COLOR_BIT`, then it **must** not include any of `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`
- `aspectMask` **must** not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`

## Valid Usage (Implicit)

- `aspectMask` **must** be a valid combination of `VkImageAspectFlagBits` values
- `aspectMask` **must** not be `0`

Bits which **can** be set in an aspect mask to specify aspects of an image for purposes such as identifying a subresource, are:

```
typedef enum VkImageAspectFlagBits {
    VK_IMAGE_ASPECT_COLOR_BIT = 0x00000001,
    VK_IMAGE_ASPECT_DEPTH_BIT = 0x00000002,
    VK_IMAGE_ASPECT_STENCIL_BIT = 0x00000004,
    VK_IMAGE_ASPECT_METADATA_BIT = 0x00000008,
    VK_IMAGE_ASPECT_PLANE_0_BIT = 0x00000010,
    VK_IMAGE_ASPECT_PLANE_1_BIT = 0x00000020,
    VK_IMAGE_ASPECT_PLANE_2_BIT = 0x00000040,
    VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT = 0x00000080,
    VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT = 0x00000100,
    VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT = 0x00000200,
    VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT = 0x00000400,
    VK_IMAGE_ASPECT_PLANE_0_BIT_KHR = VK_IMAGE_ASPECT_PLANE_0_BIT,
    VK_IMAGE_ASPECT_PLANE_1_BIT_KHR = VK_IMAGE_ASPECT_PLANE_1_BIT,
    VK_IMAGE_ASPECT_PLANE_2_BIT_KHR = VK_IMAGE_ASPECT_PLANE_2_BIT,
    VK_IMAGE_ASPECT_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkImageAspectFlagBits;
```

- `VK_IMAGE_ASPECT_COLOR_BIT` specifies the color aspect.
- `VK_IMAGE_ASPECT_DEPTH_BIT` specifies the depth aspect.
- `VK_IMAGE_ASPECT_STENCIL_BIT` specifies the stencil aspect.
- `VK_IMAGE_ASPECT_METADATA_BIT` specifies the metadata aspect, used for sparse `sparse resource` operations.
- `VK_IMAGE_ASPECT_PLANE_0_BIT` specifies plane 0 of a *multi-planar* image format.
- `VK_IMAGE_ASPECT_PLANE_1_BIT` specifies plane 1 of a *multi-planar* image format.
- `VK_IMAGE_ASPECT_PLANE_2_BIT` specifies plane 2 of a *multi-planar* image format.

- `VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT` specifies *memory plane* 0.
- `VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT` specifies *memory plane* 1.
- `VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT` specifies *memory plane* 2.
- `VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT` specifies *memory plane* 3.

```
typedef VkFlags VkImageAspectFlags;
```

`VkImageAspectFlags` is a bitmask type for setting a mask of zero or more `VkImageAspectFlagBits`.

The `VkComponentMapping` structure is defined as:

```
typedef struct VkComponentMapping {
    VkComponentSwizzle r;
    VkComponentSwizzle g;
    VkComponentSwizzle b;
    VkComponentSwizzle a;
} VkComponentMapping;
```

- `r` is a `VkComponentSwizzle` specifying the component value placed in the R component of the output vector.
- `g` is a `VkComponentSwizzle` specifying the component value placed in the G component of the output vector.
- `b` is a `VkComponentSwizzle` specifying the component value placed in the B component of the output vector.
- `a` is a `VkComponentSwizzle` specifying the component value placed in the A component of the output vector.

### Valid Usage (Implicit)

- `r` **must** be a valid `VkComponentSwizzle` value
- `g` **must** be a valid `VkComponentSwizzle` value
- `b` **must** be a valid `VkComponentSwizzle` value
- `a` **must** be a valid `VkComponentSwizzle` value

Possible values of the members of `VkComponentMapping`, specifying the component values placed in each component of the output vector, are:

```

typedef enum VkComponentSwizzle {
    VK_COMPONENT_SWIZZLE_IDENTITY = 0,
    VK_COMPONENT_SWIZZLE_ZERO = 1,
    VK_COMPONENT_SWIZZLE_ONE = 2,
    VK_COMPONENT_SWIZZLE_R = 3,
    VK_COMPONENT_SWIZZLE_G = 4,
    VK_COMPONENT_SWIZZLE_B = 5,
    VK_COMPONENT_SWIZZLE_A = 6,
    VK_COMPONENT_SWIZZLE_MAX_ENUM = 0x7FFFFFFF
} VkComponentSwizzle;

```

- **VK\_COMPONENT\_SWIZZLE\_IDENTITY** specifies that the component is set to the identity swizzle.
- **VK\_COMPONENT\_SWIZZLE\_ZERO** specifies that the component is set to zero.
- **VK\_COMPONENT\_SWIZZLE\_ONE** specifies that the component is set to either 1 or 1.0, depending on whether the type of the image view format is integer or floating-point respectively, as determined by the [Format Definition](#) section for each [VkFormat](#).
- **VK\_COMPONENT\_SWIZZLE\_R** specifies that the component is set to the value of the R component of the image.
- **VK\_COMPONENT\_SWIZZLE\_G** specifies that the component is set to the value of the G component of the image.
- **VK\_COMPONENT\_SWIZZLE\_B** specifies that the component is set to the value of the B component of the image.
- **VK\_COMPONENT\_SWIZZLE\_A** specifies that the component is set to the value of the A component of the image.

Setting the identity swizzle on a component is equivalent to setting the identity mapping on that component. That is:

*Table 16. Component Mappings Equivalent To VK\_COMPONENT\_SWIZZLE\_IDENTITY*

Component	Identity Mapping
components.r	VK_COMPONENT_SWIZZLE_R
components.g	VK_COMPONENT_SWIZZLE_G
components.b	VK_COMPONENT_SWIZZLE_B
components.a	VK_COMPONENT_SWIZZLE_A

If the **pNext** list includes a [VkImageViewASTCDecodeModeEXT](#) structure, then that structure includes a parameter specifying the decode mode for image views using ASTC compressed formats.

The [VkImageViewASTCDecodeModeEXT](#) structure is defined as:

```

typedef struct VkImageViewASTCDecodeModeEXT {
    VkStructureType      sType;
    const void*        pNext;
    VkFormat            decodeMode;
} VkImageViewASTCDecodeModeEXT;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `decodeMode` is the intermediate format used to decode ASTC compressed formats.

## Valid Usage

- `decodeMode` **must** be one of `VK_FORMAT_R16G16B16A16_SFLOAT`, `VK_FORMAT_R8G8B8A8_UNORM`, or `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32`
- If the `decodeModeSharedExponent` feature is not enabled, `decodeMode` **must** not be `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32`
- If `decodeMode` is `VK_FORMAT_R8G8B8A8_UNORM` the image view **must** not include blocks using any of the ASTC HDR modes
- `format` of the image view **must** be one of `VK_FORMAT_ASTC_4x4_UNORM_BLOCK`, `VK_FORMAT_ASTC_4x4_SRGB_BLOCK`, `VK_FORMAT_ASTC_5x4_SRGB_BLOCK`, `VK_FORMAT_ASTC_5x5_SRGB_BLOCK`, `VK_FORMAT_ASTC_6x5_SRGB_BLOCK`, `VK_FORMAT_ASTC_6x6_SRGB_BLOCK`, `VK_FORMAT_ASTC_8x5_SRGB_BLOCK`, `VK_FORMAT_ASTC_8x6_SRGB_BLOCK`, `VK_FORMAT_ASTC_8x8_SRGB_BLOCK`, `VK_FORMAT_ASTC_10x5_SRGB_BLOCK`, `VK_FORMAT_ASTC_10x6_SRGB_BLOCK`, `VK_FORMAT_ASTC_10x8_SRGB_BLOCK`, `VK_FORMAT_ASTC_10x10_SRGB_BLOCK`, `VK_FORMAT_ASTC_12x10_SRGB_BLOCK`, `VK_FORMAT_ASTC_12x12_SRGB_BLOCK`, `VK_FORMAT_ASTC_4x4_UNORM_BLOCK`, `VK_FORMAT_ASTC_5x4_UNORM_BLOCK`, `VK_FORMAT_ASTC_5x5_UNORM_BLOCK`, `VK_FORMAT_ASTC_6x5_UNORM_BLOCK`, `VK_FORMAT_ASTC_6x6_UNORM_BLOCK`, `VK_FORMAT_ASTC_8x5_UNORM_BLOCK`, `VK_FORMAT_ASTC_8x6_UNORM_BLOCK`, `VK_FORMAT_ASTC_8x8_UNORM_BLOCK`, `VK_FORMAT_ASTC_10x5_UNORM_BLOCK`, `VK_FORMAT_ASTC_10x6_UNORM_BLOCK`, `VK_FORMAT_ASTC_10x8_UNORM_BLOCK`, `VK_FORMAT_ASTC_10x10_UNORM_BLOCK`, `VK_FORMAT_ASTC_12x10_UNORM_BLOCK`, `VK_FORMAT_ASTC_12x12_UNORM_BLOCK`, or

If `format` uses sRGB encoding then the `decodeMode` has no effect.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT`
- `decodeMode` **must** be a valid `VkFormat` value

To destroy an image view, call:

```
void vkDestroyImageView(  
    VkDevice device,  
    VkImageView imageView,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the image view.
- `imageView` is the image view to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `imageView` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `imageView` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `imageView` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `imageView` is not `VK_NULL_HANDLE`, `imageView` **must** be a valid `VkImageView` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `imageView` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `imageView` **must** be externally synchronized

To get the handle for an image view, call:

```
uint32_t vkGetImageViewHandleNVX(  
    VkDevice device,  
    const VkImageViewHandleInfoNVX* pInfo);
```

- `device` is the logical device that owns the image view.
- `pInfo` describes the image view to query and type of handle.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkImageViewHandleInfoNVX` structure

The `VkImageViewHandleInfoNVX` structure is defined as:

```
typedef struct VkImageViewHandleInfoNVX {
    VkStructureType    sType;
    const void*        pNext;
    VkImageView        imageView;
    VkDescriptorType   descriptorType;
    VkSampler          sampler;
} VkImageViewHandleInfoNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `imageView` is the image view to query.
- `descriptorType` is the type of descriptor for which to query a handle.
- `sampler` is the sampler to combine with the image view when generating the handle.

## Valid Usage

- `descriptorType` **must** be `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`
- `sampler` **must** be a valid `VkSampler` if `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the image that `imageView` was created from **must** have been created with the `VK_IMAGE_USAGE_SAMPLED_BIT` usage bit set
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, the image that `imageView` was created from **must** have been created with the `VK_IMAGE_USAGE_STORAGE_BIT` usage bit set

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_VIEW_HANDLE_INFO_NVX`
- `pNext` **must** be `NULL`
- `imageView` **must** be a valid `VkImageView` handle
- `descriptorType` **must** be a valid `VkDescriptorType` value
- If `sampler` is not `VK_NULL_HANDLE`, `sampler` **must** be a valid `VkSampler` handle
- Both of `imageView`, and `sampler` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

### 11.5.1. Image View Format Features

Valid usage of a `VkImageView` **may** be constrained by the image view's format features, defined below. Such constraints are documented in the affected valid usage statement.

- If the view's image was created with `VK_IMAGE_TILING_LINEAR`, then the image view's set of *format features* is the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageViewCreateInfo::format`.
- If the view's image was created with `VK_IMAGE_TILING_OPTIMAL`, but without an `external format`, then the image view's set of *format features* is the value of `VkFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageViewCreateInfo::format`.
- If the view's image was created with an `external format`, then the image views's set of *format features* is the value of `VkAndroidHardwareBufferFormatPropertiesANDROID::formatFeatures` found by calling `vkGetAndroidHardwareBufferPropertiesANDROID` on the Android hardware buffer that was imported to the `VkDeviceMemory` to which the image is bound.
- If the view's image was created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then:
  - The image's DRM format modifier is the value of `VkImageDrmFormatModifierListCreateInfoEXT::drmFormatModifier` found by calling `vkGetImageDrmFormatModifierPropertiesEXT`.
  - Let `VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties` be the array found by calling `vkGetPhysicalDeviceFormatProperties2` on the same `format` as `VkImageViewCreateInfo::format`.
  - Let `VkDrmFormatModifierPropertiesEXT prop` be the array element whose `drmFormatModifier` member is the value of the image's DRM format modifier.
  - Then the image view's set of *format features* is the value of `prop::drmFormatModifierTilingFeatures`.

## 11.6. Resource Memory Association

Resources are initially created as *virtual allocations* with no backing memory. Device memory is allocated separately (see [Device Memory](#)) and then associated with the resource. This association is

done differently for sparse and non-sparse resources.

Resources created with any of the sparse creation flags are considered sparse resources. Resources created without these flags are non-sparse. The details on resource memory association for sparse resources is described in [Sparse Resources](#).

Non-sparse resources **must** be bound completely and contiguously to a single `VkDeviceMemory` object before the resource is passed as a parameter to any of the following operations:

- creating image or buffer views
- updating descriptor sets
- recording commands in a command buffer

Once bound, the memory binding is immutable for the lifetime of the resource.

In a logical device representing more than one physical device, buffer and image resources exist on all physical devices but **can** be bound to memory differently on each. Each such replicated resource is an *instance* of the resource. For sparse resources, each instance **can** be bound to memory arbitrarily differently. For non-sparse resources, each instance **can** either be bound to the local or a peer instance of the memory, or for images **can** be bound to rectangular regions from the local and/or peer instances. When a resource is used in a descriptor set, each physical device interprets the descriptor according to its own instance's binding to memory.

*Note*



There are no new copy commands to transfer data between physical devices. Instead, an application **can** create a resource with a peer mapping and use it as the source or destination of a transfer command executed by a single physical device to copy the data from one physical device to another.

To determine the memory requirements for a buffer resource, call:

```
void vkGetBufferMemoryRequirements(  
    VkDevice device,  
    VkBuffer buffer,  
    VkMemoryRequirements* pMemoryRequirements);
```

- `device` is the logical device that owns the buffer.
- `buffer` is the buffer to query.
- `pMemoryRequirements` is a pointer to a `VkMemoryRequirements` structure in which the memory requirements of the buffer object are returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `pMemoryRequirements` **must** be a valid pointer to a `VkMemoryRequirements` structure
- `buffer` **must** have been created, allocated, or retrieved from `device`

To determine the memory requirements for an image resource which is not created with the `VK_IMAGE_CREATE_DISJOINT_BIT` flag set, call:

```
void vkGetImageMemoryRequirements(  
    VkDevice device,  
    VkImage image,  
    VkMemoryRequirements* pMemoryRequirements);
```

- `device` is the logical device that owns the image.
- `image` is the image to query.
- `pMemoryRequirements` is a pointer to a `VkMemoryRequirements` structure in which the memory requirements of the image object are returned.

## Valid Usage

- `image` **must** not have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` flag set

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `image` **must** be a valid `VkImage` handle
- `pMemoryRequirements` **must** be a valid pointer to a `VkMemoryRequirements` structure
- `image` **must** have been created, allocated, or retrieved from `device`

The `VkMemoryRequirements` structure is defined as:

```
typedef struct VkMemoryRequirements {  
    VkDeviceSize size;  
    VkDeviceSize alignment;  
    uint32_t memoryTypeBits;  
} VkMemoryRequirements;
```

- `size` is the size, in bytes, of the memory allocation **required** for the resource.

- `alignment` is the alignment, in bytes, of the offset within the allocation **required** for the resource.
- `memoryTypeBits` is a bitmask and contains one bit set for every supported memory type for the resource. Bit `i` is set if and only if the memory type `i` in the `VkPhysicalDeviceMemoryProperties` structure for the physical device is supported for the resource.

The precise size of images that will be bound to external Android hardware buffer memory is unknown until the memory has been imported or allocated, so applications **must** not call `vkGetImageMemoryRequirements` with such an image before it has been bound to memory. When importing Android hardware buffer memory, the `allocationSize` **can** be determined by calling `vkGetAndroidHardwareBufferPropertiesANDROID`. When allocating new memory for an image that **can** be exported to an Android hardware buffer, the memory's `allocationSize` **must** be zero; the actual size will be determined by the dedicated image's parameters. After the memory has been allocated, the amount of space allocated from the memory's heap **can** be obtained by getting the image's memory requirements or by calling `vkGetAndroidHardwareBufferPropertiesANDROID` with the Android hardware buffer exported from the memory.

The implementation guarantees certain properties about the memory requirements returned by `vkGetBufferMemoryRequirements` and `vkGetImageMemoryRequirements`:

- The `memoryTypeBits` member always contains at least one bit set.
- If `buffer` is a `VkBuffer` not created with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` bit set, or if `image` is `linear` image, then the `memoryTypeBits` member always contains at least one bit set corresponding to a `VkMemoryType` with a `propertyFlags` that has both the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` bit and the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` bit set. In other words, mappable coherent memory **can** always be attached to these objects.
- If `buffer` was created with `VkExternalMemoryBufferCreateInfo::handleTypes` set to `0` or `image` was created with `VkExternalMemoryImageCreateInfo::handleTypes` set to `0`, the `memoryTypeBits` member always contains at least one bit set corresponding to a `VkMemoryType` with a `propertyFlags` that has the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` bit set.
- The `memoryTypeBits` member is identical for all `VkBuffer` objects created with the same value for the `flags` and `usage` members in the `VkBufferCreateInfo` structure and the `handleTypes` member of the `VkExternalMemoryBufferCreateInfo` structure passed to `vkCreateBuffer`. Further, if `usage1` and `usage2` of type `VkBufferUsageFlags` are such that the bits set in `usage2` are a subset of the bits set in `usage1`, and they have the same `flags` and `VkExternalMemoryBufferCreateInfo::handleTypes`, then the bits set in `memoryTypeBits` returned for `usage1` **must** be a subset of the bits set in `memoryTypeBits` returned for `usage2`, for all values of `flags`.
- The `alignment` member is a power of two.
- The `alignment` member is identical for all `VkBuffer` objects created with the same combination of values for the `usage` and `flags` members in the `VkBufferCreateInfo` structure passed to `vkCreateBuffer`.
- The `alignment` member satisfies the buffer descriptor offset alignment requirements associated with the `VkBuffer`'s `usage`:
  - If `usage` included `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, `alignment` **must** be an integer multiple of

`VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`.

- If `usage` included `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT`, `alignment` **must** be an integer multiple of `VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment`.
- If `usage` included `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT`, `alignment` **must** be an integer multiple of `VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment`.
- For images created with a color format, the `memoryTypeBits` member is identical for all `VkImage` objects created with the same combination of values for the `tiling` member, the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` bit of the `flags` member, the `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` bit of the `flags` member, `handleTypes` member of `VkExternalMemoryImageCreateInfo`, and the `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` of the `usage` member in the `VkImageCreateInfo` structure passed to `vkCreateImage`.
- For images created with a depth/stencil format, the `memoryTypeBits` member is identical for all `VkImage` objects created with the same combination of values for the `format` member, the `tiling` member, the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` bit of the `flags` member, the `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` bit of the `flags` member, `handleTypes` member of `VkExternalMemoryImageCreateInfo`, and the `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` of the `usage` member in the `VkImageCreateInfo` structure passed to `vkCreateImage`.
- If the memory requirements are for a `VkImage`, the `memoryTypeBits` member **must** not refer to a `VkMemoryType` with a `propertyFlags` that has the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit set if the `image` did not have `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` bit set in the `usage` member of the `VkImageCreateInfo` structure passed to `vkCreateImage`.
- If the memory requirements are for a `VkBuffer`, the `memoryTypeBits` member **must** not refer to a `VkMemoryType` with a `propertyFlags` that has the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit set.

*Note*



The implication of this requirement is that lazily allocated memory is disallowed for buffers in all cases.

- The `size` member is identical for all `VkBuffer` objects created with the same combination of creation parameters specified in `VkBufferCreateInfo` and its `pNext` chain.
- The `size` member is identical for all `VkImage` objects created with the same combination of creation parameters specified in `VkImageCreateInfo` and its `pNext` chain.

*Note*



This, however, does not imply that they interpret the contents of the bound memory identically with each other. That additional guarantee, however, **can** be explicitly requested using `VK_IMAGE_CREATE_ALIAS_BIT`.

To determine the memory requirements for a buffer resource, call:

```
void vkGetBufferMemoryRequirements2(
    VkDevice device,
    const VkBufferMemoryRequirementsInfo2* pInfo,
    VkMemoryRequirements2* pMemoryRequirements);
```

or the equivalent command

```
void vkGetBufferMemoryRequirements2KHR(
    VkDevice device,
    const VkBufferMemoryRequirementsInfo2* pInfo,
    VkMemoryRequirements2* pMemoryRequirements);
```

- **device** is the logical device that owns the buffer.
- **pInfo** is a pointer to a `VkBufferMemoryRequirementsInfo2` structure containing parameters required for the memory requirements query.
- **pMemoryRequirements** is a pointer to a `VkMemoryRequirements2` structure in which the memory requirements of the buffer object are returned.

### Valid Usage (Implicit)

- **device** **must** be a valid `VkDevice` handle
- **pInfo** **must** be a valid pointer to a valid `VkBufferMemoryRequirementsInfo2` structure
- **pMemoryRequirements** **must** be a valid pointer to a `VkMemoryRequirements2` structure

The `VkBufferMemoryRequirementsInfo2` structure is defined as:

```
typedef struct VkBufferMemoryRequirementsInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkBuffer buffer;
} VkBufferMemoryRequirementsInfo2;
```

or the equivalent

```
typedef VkBufferMemoryRequirementsInfo2 VkBufferMemoryRequirementsInfo2KHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **buffer** is the buffer to query.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2`
- `pNext` **must** be `NULL`
- `buffer` **must** be a valid `VkBuffer` handle

To determine the memory requirements for an image resource, call:

```
void vkGetImageMemoryRequirements2(
    VkDevice                                     device,
    const VkImageMemoryRequirementsInfo2*        pInfo,
    VkMemoryRequirements2*                      pMemoryRequirements);
```

or the equivalent command

```
void vkGetImageMemoryRequirements2KHR(
    VkDevice                                     device,
    const VkImageMemoryRequirementsInfo2*       pInfo,
    VkMemoryRequirements2*                     pMemoryRequirements);
```

- `device` is the logical device that owns the image.
- `pInfo` is a pointer to a `VkImageMemoryRequirementsInfo2` structure containing parameters required for the memory requirements query.
- `pMemoryRequirements` is a pointer to a `VkMemoryRequirements2` structure in which the memory requirements of the image object are returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkImageMemoryRequirementsInfo2` structure
- `pMemoryRequirements` **must** be a valid pointer to a `VkMemoryRequirements2` structure

The `VkImageMemoryRequirementsInfo2` structure is defined as:

```
typedef struct VkImageMemoryRequirementsInfo2 {
    VkStructureType    sType;
    const void*       pNext;
    VkImage           image;
} VkImageMemoryRequirementsInfo2;
```

or the equivalent

```
typedef VkImageMemoryRequirementsInfo2 VkImageMemoryRequirementsInfo2KHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **image** is the image to query.

## Valid Usage

- If **image** was created with a *multi-planar* format and the **VK\_IMAGE\_CREATE\_DISJOINT\_BIT** flag, there **must** be a **VkImagePlaneMemoryRequirementsInfo** in the **pNext** chain of the **VkImageMemoryRequirementsInfo2** structure
- If **image** was created with **VK\_IMAGE\_CREATE\_DISJOINT\_BIT** and with **VK\_IMAGE\_TILING\_DRM\_FORMAT\_MODIFIER\_EXT**, then there **must** be a **VkImagePlaneMemoryRequirementsInfo** in the **pNext** chain of the **VkImageMemoryRequirementsInfo2** structure
- If **image** was not created with the **VK\_IMAGE\_CREATE\_DISJOINT\_BIT** flag, there **must** not be a **VkImagePlaneMemoryRequirementsInfo** in the **pNext** chain of the **VkImageMemoryRequirementsInfo2** structure
- If **image** was created with a single-plane format and with any **tiling** other than **VK\_IMAGE\_TILING\_DRM\_FORMAT\_MODIFIER\_EXT**, then there **must** not be a **VkImagePlaneMemoryRequirementsInfo** in the **pNext** chain of the **VkImageMemoryRequirementsInfo2** structure
- If **image** was created with the **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_ANDROID\_HARDWARE\_BUFFER\_BIT\_ANDROID** external memory handle type, then **image** **must** be bound to memory.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_IMAGE\_MEMORY\_REQUIREMENTS\_INFO\_2**
- **pNext** **must** be **NULL** or a pointer to a valid instance of **VkImagePlaneMemoryRequirementsInfo**
- **image** **must** be a valid **VkImage** handle

To determine the memory requirements for a plane of a disjoint image, add a **VkImagePlaneMemoryRequirementsInfo** to the **pNext** chain of the **VkImageMemoryRequirementsInfo2** structure.

The **VkImagePlaneMemoryRequirementsInfo** structure is defined as:

```
typedef struct VkImagePlaneMemoryRequirementsInfo {
    VkStructureType      sType;
    const void*          pNext;
    VkImageAspectFlagBits planeAspect;
} VkImagePlaneMemoryRequirementsInfo;
```

or the equivalent

```
typedef VkImagePlaneMemoryRequirementsInfo VkImagePlaneMemoryRequirementsInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `planeAspect` is the aspect corresponding to the image plane to query.

## Valid Usage

- If the image's tiling is `VK_IMAGE_TILING_LINEAR` or `VK_IMAGE_TILING_OPTIMAL`, then `planeAspect` **must** be a single valid *format plane* for the image. (That is, for a two-plane image `planeAspect` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`, and for a three-plane image `planeAspect` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or `VK_IMAGE_ASPECT_PLANE_2_BIT`).
- If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `planeAspect` **must** be a single valid *memory plane* for the image. (That is, `aspectMask` **must** specify a plane index that is less than the `drmFormatModifierPlaneCount` associated with the image's `format` and `drmFormatModifier`.)

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO`
- `planeAspect` **must** be a valid `VkImageAspectFlagBits` value

The `VkMemoryRequirements2` structure is defined as:

```
typedef struct VkMemoryRequirements2 {
    VkStructureType      sType;
    void*               pNext;
    VkMemoryRequirements memoryRequirements;
} VkMemoryRequirements2;
```

or the equivalent

```
typedef VkMemoryRequirements2 VkMemoryRequirements2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memoryRequirements` is a `VkMemoryRequirements` structure describing the memory requirements of the resource.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkMemoryDedicatedRequirements`

To determine the dedicated allocation requirements of a buffer or image resource, add a `VkMemoryDedicatedRequirements` structure to the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of `vkGetBufferMemoryRequirements2` or `vkGetImageMemoryRequirements2`.

The `VkMemoryDedicatedRequirements` structure is defined as:

```
typedef struct VkMemoryDedicatedRequirements {  
    VkStructureType    sType;  
    void*              pNext;  
    VkBool32            prefersDedicatedAllocation;  
    VkBool32            requiresDedicatedAllocation;  
} VkMemoryDedicatedRequirements;
```

or the equivalent

```
typedef VkMemoryDedicatedRequirements VkMemoryDedicatedRequirementsKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `prefersDedicatedAllocation` specifies that the implementation would prefer a dedicated allocation for this resource. The application is still free to suballocate the resource but it **may** get better performance if a dedicated allocation is used.
- `requiresDedicatedAllocation` specifies that a dedicated allocation is required for this resource.

When the implementation sets `requiresDedicatedAllocation` to `VK_TRUE`, it **must** also set `prefersDedicatedAllocation` to `VK_TRUE`.

If the `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of a `vkGetBufferMemoryRequirements2` call, `requiresDedicatedAllocation` **may** be `VK_TRUE` under one of the

following conditions:

- The `pNext` chain of `VkBufferCreateInfo` for the call to `vkCreateBuffer` used to create the buffer being queried contained an instance of `VkExternalMemoryBufferCreateInfo`, and any of the handle types specified in `VkExternalMemoryBufferCreateInfo::handleTypes` requires dedicated allocation, as reported by `vkGetPhysicalDeviceExternalBufferProperties` in `VkExternalBufferProperties::externalMemoryProperties::externalMemoryFeatures`, the `requiresDedicatedAllocation` field will be set to `VK_TRUE`.

In all other cases, `requiresDedicatedAllocation` **must** be set to `VK_FALSE` by the implementation whenever a `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed to a call to `vkGetBufferMemoryRequirements2`.

If the `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of a `vkGetBufferMemoryRequirements2` call and `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` was set in `VkBufferCreateInfo::flags` when `buffer` was created then the implementation **must** set both `prefersDedicatedAllocation` and `requiresDedicatedAllocation` to `VK_FALSE`.

If the `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of a `vkGetImageMemoryRequirements2` call, `requiresDedicatedAllocation` **may** be `VK_TRUE` under one of the following conditions:

- The `pNext` chain of `VkImageCreateInfo` for the call to `vkCreateImage` used to create the image being queried contained an instance of `VkExternalMemoryImageCreateInfo`, and any of the handle types specified in `VkExternalMemoryImageCreateInfo::handleTypes` requires dedicated allocation, as reported by `vkGetPhysicalDeviceImageFormatProperties2` in `VkExternalImageFormatProperties::externalMemoryProperties::externalMemoryFeatures`, the `requiresDedicatedAllocation` field will be set to `VK_TRUE`.

In all other cases, `requiresDedicatedAllocation` **must** be set to `VK_FALSE` by the implementation whenever a `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed to a call to `vkGetImageMemoryRequirements2`.

If the `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of a `vkGetImageMemoryRequirements2` call and `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` was set in `VkImageCreateInfo::flags` when `image` was created then the implementation **must** set both `prefersDedicatedAllocation` and `requiresDedicatedAllocation` to `VK_FALSE`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS`

To attach memory to a buffer object, call:

```
VkResult vkBindBufferMemory(  
    VkDevice device,  
    VkBuffer buffer,  
    VkDeviceMemory memory,  
    VkDeviceSize memoryOffset);
```

- **device** is the logical device that owns the buffer and memory.
- **buffer** is the buffer to be attached to memory.
- **memory** is a [VkDeviceMemory](#) object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of **memory** which is to be bound to the buffer. The number of bytes returned in the [VkMemoryRequirements::size](#) member in **memory**, starting from **memoryOffset** bytes, will be bound to the specified buffer.

[vkBindBufferMemory](#) is equivalent to passing the same parameters through [VkBindBufferMemoryInfo](#) to [vkBindBufferMemory2](#).

## Valid Usage

- `buffer` **must** not already be backed by a memory object
- `buffer` **must** not have been created with any sparse memory binding flags
- `memoryOffset` **must** be less than the size of `memory`
- `memory` **must** have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetBufferMemoryRequirements` with `buffer`
- `memoryOffset` **must** be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetBufferMemoryRequirements` with `buffer`
- The `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetBufferMemoryRequirements` with `buffer` **must** be less than or equal to the size of `memory` minus `memoryOffset`
- If `buffer` requires a dedicated allocation(as reported by `vkGetBufferMemoryRequirements2` `VkMemoryDedicatedRequirements::requiresDedicatedAllocation` for `buffer`), `memory` **must** have been created with `VkMemoryDedicatedAllocateInfo::buffer` equal to `buffer`
- If the `VkMemoryAllocateInfo` provided when `memory` was allocated included an instance of `VkMemoryDedicatedAllocateInfo` in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::buffer` was not `VK_NULL_HANDLE`, then `buffer` **must** equal `VkMemoryDedicatedAllocateInfo::buffer`, and `memoryOffset` **must** be zero.
- If `buffer` was created with the `VK_BUFFER_CREATE_PROTECTED_BIT` bit set, the buffer **must** be bound to a memory object allocated with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`
- If `buffer` was created with the `VK_BUFFER_CREATE_PROTECTED_BIT` bit not set, the buffer **must** not be bound to a memory object created with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`
- If `buffer` was created with `VkDedicatedAllocationBufferCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`, `memory` **must** have been created with `VkDedicatedAllocationMemoryAllocateInfoNV::buffer` equal to a buffer handle created with identical creation parameters to `buffer` and `memoryOffset` **must** be zero
- If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not `0`, it **must** include at least one of the handles set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created
- If `memory` was created by a memory import operation, the external handle type of the imported memory **must** also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created
- If the `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR::bufferDeviceAddress` feature is enabled and `buffer` was created with the `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR` bit set, `memory` **must** have been allocated with the `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR` bit set

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- `buffer` **must** have been created, allocated, or retrieved from `device`
- `memory` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `buffer` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR`

To attach memory to buffer objects for one or more buffers at a time, call:

```
VkResult vkBindBufferMemory2(  
    VkDevice                                     device,  
    uint32_t                                      bindInfoCount,  
    const VkBindBufferMemoryInfo*                  pBindInfos);
```

or the equivalent command

```
VkResult vkBindBufferMemory2KHR(  
    VkDevice                                     device,  
    uint32_t                                      bindInfoCount,  
    const VkBindBufferMemoryInfo*                  pBindInfos);
```

- `device` is the logical device that owns the buffers and memory.
- `bindInfoCount` is the number of elements in `pBindInfos`.
- `pBindInfos` is a pointer to an array of `bindInfoCount` `VkBindBufferMemoryInfo` structures describing buffers and memory to bind.

On some implementations, it **may** be more efficient to batch memory bindings into a single command.

## Valid Usage (Implicit)

- **device** **must** be a valid `VkDevice` handle
- **pBindInfos** **must** be a valid pointer to an array of `bindInfoCount` valid `VkBindBufferMemoryInfo` structures
- **bindInfoCount** **must** be greater than `0`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR`

`VkBindBufferMemoryInfo` contains members corresponding to the parameters of `vkBindBufferMemory`.

The `VkBindBufferMemoryInfo` structure is defined as:

```
typedef struct VkBindBufferMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkBuffer buffer;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
} VkBindBufferMemoryInfo;
```

or the equivalent

```
typedef VkBindBufferMemoryInfo VkBindBufferMemoryInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **buffer** is the buffer to be attached to memory.
- **memory** is a `VkDeviceMemory` object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of `memory` which is to be bound to the buffer. The

number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified buffer.

## Valid Usage

- `buffer` **must** not already be backed by a memory object
- `buffer` **must** not have been created with any sparse memory binding flags
- `memoryOffset` **must** be less than the size of `memory`
- `memory` **must** have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetBufferMemoryRequirements` with `buffer`
- `memoryOffset` **must** be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetBufferMemoryRequirements` with `buffer`
- The `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetBufferMemoryRequirements` with `buffer` **must** be less than or equal to the size of `memory` minus `memoryOffset`
- If `buffer` requires a dedicated allocation (as reported by `vkGetBufferMemoryRequirements2` in `VkMemoryDedicatedRequirements::requiresDedicatedAllocation` for `buffer`), `memory` **must** have been created with `VkMemoryDedicatedAllocateInfo::buffer` equal to `buffer` and `memoryOffset` **must** be zero
- If the `VkMemoryAllocateInfo` provided when `memory` was allocated included an instance of `VkMemoryDedicatedAllocateInfo` in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::buffer` was not `VK_NULL_HANDLE`, then `buffer` **must** equal `VkMemoryDedicatedAllocateInfo::buffer` and `memoryOffset` **must** be zero.
- If `buffer` was created with `VkDedicatedAllocationBufferCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`, `memory` **must** have been created with `VkDedicatedAllocationMemoryAllocateInfoNV::buffer` equal to `buffer` and `memoryOffset` **must** be zero
- If the `pNext` chain includes `VkBIndBufferMemoryDeviceGroupInfo`, all instances of `memory` specified by `VkBIndBufferMemoryDeviceGroupInfo::pDeviceIndices` **must** have been allocated
- If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not `0`, it **must** include at least one of the handles set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created
- If `memory` was created by a memory import operation, the external handle type of the imported memory **must** also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO`
- `pNext` must be `NULL` or a pointer to a valid instance of `VkBindBufferMemoryDeviceGroupInfo`
- `buffer` must be a valid `VkBuffer` handle
- `memory` must be a valid `VkDeviceMemory` handle
- Both of `buffer`, and `memory` must have been created, allocated, or retrieved from the same `VkDevice`

```
typedef struct VkBindBufferMemoryDeviceGroupInfo {  
    VkStructureType sType;  
    const void* pNext;  
    uint32_t deviceIndexCount;  
    const uint32_t* pDeviceIndices;  
} VkBindBufferMemoryDeviceGroupInfo;
```

or the equivalent

```
typedef VkBindBufferMemoryDeviceGroupInfo VkBindBufferMemoryDeviceGroupInfoKHR;
```

If the `pNext` list of `VkBindBufferMemoryInfo` includes a `VkBindBufferMemoryDeviceGroupInfo` structure, then that structure determines how memory is bound to buffers across multiple devices in a device group.

The `VkBindBufferMemoryDeviceGroupInfo` structure is defined as:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `deviceIndexCount` is the number of elements in `pDeviceIndices`.
- `pDeviceIndices` is a pointer to an array of device indices.

If `deviceIndexCount` is greater than zero, then on device index `i` the buffer is attached to the instance of `memory` on the physical device with device index `pDeviceIndices[i]`.

If `deviceIndexCount` is zero and `memory` comes from a memory heap with the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains consecutive indices from zero to the number of physical devices in the logical device, minus one. In other words, by default each physical device attaches to its own instance of `memory`.

If `deviceIndexCount` is zero and `memory` comes from a memory heap without the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains an array of zeros. In other words, by default each physical device attaches to instance zero.

## Valid Usage

- `deviceIndexCount` **must** either be zero or equal to the number of physical devices in the logical device
- All elements of `pDeviceIndices` **must** be valid device indices

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO`
- If `deviceIndexCount` is not `0`, `pDeviceIndices` **must** be a valid pointer to an array of `deviceIndexCount` `uint32_t` values

To attach memory to a `VkImage` object created without the `VK_IMAGE_CREATE_DISJOINT_BIT` set, call:

```
VkResult vkBindImageMemory(  
    VkDevice                          device,  
    VkImage                           image,  
    VkDeviceMemory                    memory,  
    VkDeviceSize                     memoryOffset);
```

- `device` is the logical device that owns the image and memory.
- `image` is the image.
- `memory` is the `VkDeviceMemory` object describing the device memory to attach.
- `memoryOffset` is the start offset of the region of `memory` which is to be bound to the image. The number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified image.

`vkBindImageMemory` is equivalent to passing the same parameters through `VkBindImageMemoryInfo` to `vkBindImageMemory2`.

## Valid Usage

- `image` **must** not have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` set.
- `image` **must** not already be backed by a memory object
- `image` **must** not have been created with any sparse memory binding flags
- `memoryOffset` **must** be less than the size of `memory`
- `memory` **must** have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with `image`
- `memoryOffset` **must** be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with `image`
- The `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with `image` **must** be less than or equal to the size of `memory` minus `memoryOffset`
- If `image` requires a dedicated allocation (as reported by `vkGetImageMemoryRequirements2` `VkMemoryDedicatedRequirements::requiresDedicatedAllocation` for `image`), `memory` **must** have been created with `VkMemoryDedicatedAllocateInfo::image` equal to `image`
- If the dedicated allocation image aliasing feature is not enabled, and the `VkMemoryAllocateInfo` provided when `memory` was allocated included an instance of `VkMemoryDedicatedAllocateInfo` in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::image` was not `VK_NULL_HANDLE`, then `image` **must** equal `VkMemoryDedicatedAllocateInfo::image` and `memoryOffset` **must** be zero.
- If the dedicated allocation image aliasing feature is enabled, and the `VkMemoryAllocateInfo` provided when `memory` was allocated included an instance of `VkMemoryDedicatedAllocateInfo` in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::image` was not `VK_NULL_HANDLE`, then `memoryOffset` **must** be zero, and `image` **must** be either equal to `VkMemoryDedicatedAllocateInfo::image` or an image that was created using the same parameters in `VkImageCreateInfo`, with the exception that `extent` and `arrayLayers` **may** differ subject to the following restrictions: every dimension in the `extent` parameter of the image being bound **must** be equal to or smaller than the original image for which the allocation was created; and the `arrayLayers` parameter of the image being bound **must** be equal to or smaller than the original image for which the allocation was created.
- If `image` was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit set, the `image` **must** be bound to a memory object allocated with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`
- If `image` was created with the `VK_IMAGE_CREATE_PROTECTED_BIT` bit not set, the `image` **must** not be bound to a memory object created with a memory type that reports `VK_MEMORY_PROPERTY_PROTECTED_BIT`
- If `image` was created with `VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`, `memory` **must** have been created with

`VkDedicatedAllocationMemoryAllocateInfoNV::image` equal to an image handle created with identical creation parameters to `image` and `memoryOffset` **must** be zero

- If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not `0`, it **must** include at least one of the handles set in `VkExternalMemoryImageCreateInfo ::handleTypes` when `image` was created
- If `memory` was created by a memory import operation, the external handle type of the imported memory **must** also have been set in `VkExternalMemoryImageCreateInfo ::handleTypes` when `image` was created

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `image` **must** be a valid `VkImage` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- `image` **must** have been created, allocated, or retrieved from `device`
- `memory` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `image` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

To attach memory to image objects for one or more images at a time, call:

```
VkResult vkBindImageMemory2(  
    VkDevice device,  
    uint32_t bindInfoCount,  
    const VkBindImageMemoryInfo* pBindInfos);
```

or the equivalent command

```

VkResult vkBindImageMemory2KHR(
    VkDevice device,
    uint32_t bindInfoCount,
    const VkBindImageMemoryInfo* pBindInfos);

```

- `device` is the logical device that owns the images and memory.
- `bindInfoCount` is the number of elements in `pBindInfos`.
- `pBindInfos` is a pointer to an array of `VkBindImageMemoryInfo` structures, describing images and memory to bind.

On some implementations, it **may** be more efficient to batch memory bindings into a single command.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pBindInfos` **must** be a valid pointer to an array of `bindInfoCount` valid `VkBindImageMemoryInfo` structures
- `bindInfoCount` **must** be greater than 0

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

`VkBindImageMemoryInfo` contains members corresponding to the parameters of `vkBindImageMemory`.

The `VkBindImageMemoryInfo` structure is defined as:

```

typedef struct VkBindImageMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
} VkBindImageMemoryInfo;

```

or the equivalent

```
typedef VkBindImageMemoryInfo VkBindImageMemoryInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **image** is the image to be attached to memory.
- **memory** is a [VkDeviceMemory](#) object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of **memory** which is to be bound to the image. The number of bytes returned in the [VkMemoryRequirements::size](#) member in **memory**, starting from **memoryOffset** bytes, will be bound to the specified image.

## Valid Usage

- `image` **must** not already be backed by a memory object
- `image` **must** not have been created with any sparse memory binding flags
- `memoryOffset` **must** be less than the size of `memory`
- If the `pNext` chain does not include an instance of the `VkBindImagePlaneMemoryInfo` structure, `memory` **must** have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with `image`
- If the `pNext` chain does not include an instance of the `VkBindImagePlaneMemoryInfo` structure, `memoryOffset` **must** be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with `image`
- If the `pNext` chain does not include an instance of the `VkBindImagePlaneMemoryInfo` structure, the difference of the size of `memory` and `memoryOffset` **must** be greater than or equal to the `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with the same `image`
- If the `pNext` chain includes an instance of the `VkBindImagePlaneMemoryInfo` structure, `image` **must** have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` bit set.
- If the `pNext` chain includes an instance of the `VkBindImagePlaneMemoryInfo` structure, `memory` **must** have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with `image` and the correct `planeAspect` for this plane in the `VkImagePlaneMemoryRequirementsInfo` structure attached to the `VkImageMemoryRequirementsInfo2`'s `pNext` chain
- If the `pNext` chain includes an instance of the `VkBindImagePlaneMemoryInfo` structure, `memoryOffset` **must** be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with `image` and the correct `planeAspect` for this plane in the `VkImagePlaneMemoryRequirementsInfo` structure attached to the `VkImageMemoryRequirementsInfo2`'s `pNext` chain
- If the `pNext` chain includes an instance of the `VkBindImagePlaneMemoryInfo` structure, the difference of the size of `memory` and `memoryOffset` **must** be greater than or equal to the `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with the same `image` and the correct `planeAspect` for this plane in the `VkImagePlaneMemoryRequirementsInfo` structure attached to the `VkImageMemoryRequirementsInfo2`'s `pNext` chain
- If `image` requires a dedicated allocation (as reported by `vkGetImageMemoryRequirements2` in `VkMemoryDedicatedRequirements::requiresDedicatedAllocation` for `image`), `memory` **must** have been created with `VkMemoryDedicatedAllocateInfo::image` equal to `image` and `memoryOffset` **must** be zero
- If the dedicated allocation image aliasing feature is not enabled, and the

`VkMemoryAllocateInfo` provided when `memory` was allocated included an instance of `VkMemoryDedicatedAllocateInfo` in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::image` was not `VK_NULL_HANDLE`, then `image must` equal `VkMemoryDedicatedAllocateInfo::image` and `memoryOffset must` be zero.

- If the dedicated allocation image aliasing feature is enabled, and the `VkMemoryAllocateInfo` provided when `memory` was allocated included an instance of `VkMemoryDedicatedAllocateInfo` in its `pNext` chain, and `VkMemoryDedicatedAllocateInfo::image` was not `VK_NULL_HANDLE`, then `memoryOffset must` be zero, and `image must` be either equal to `VkMemoryDedicatedAllocateInfo::image` or an image that was created using the same parameters in `VkImageCreateInfo`, with the exception that `extent` and `arrayLayers` **may** differ subject to the following restrictions: every dimension in the `extent` parameter of the image being bound **must** be equal to or smaller than the original image for which the allocation was created; and the `arrayLayers` parameter of the image being bound **must** be equal to or smaller than the original image for which the allocation was created.
- If `image` was created with `VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`, `memory must` have been created with `VkDedicatedAllocationMemoryAllocateInfoNV::image` equal to `image` and `memoryOffset must` be zero
- If the `pNext` chain includes `VkBindImageMemoryDeviceGroupInfo`, all instances of `memory` specified by `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` **must** have been allocated
- If the `pNext` chain includes `VkBindImageMemoryDeviceGroupInfo`, and `VkBindImageMemoryDeviceGroupInfo::splitInstanceBindRegionCount` is not zero, then `image must` have been created with the `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` bit set
- If the `pNext` chain includes `VkBindImageMemoryDeviceGroupInfo`, all elements of `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions` **must** be valid rectangles contained within the dimensions of `image`
- If the `pNext` chain includes `VkBindImageMemoryDeviceGroupInfo`, the union of the areas of all elements of `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions` that correspond to the same instance of `image` **must** cover the entire image.
- If `image` was created with a valid swapchain handle in `VkImageSwapchainCreateInfoKHR::swapchain`, then the `pNext` chain **must** include a valid instance of `VkBindImageMemorySwapchainInfoKHR` containing the same swapchain handle.
- If the `pNext` chain includes an instance of `VkBindImageMemorySwapchainInfoKHR`, `memory must` be `VK_NULL_HANDLE`
- If the `pNext` chain does not include an instance of `VkBindImageMemorySwapchainInfoKHR`, `memory must` be a valid `VkDeviceMemory` handle
- If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not `0`, it **must** include at least one of the handles set in `VkExternalMemoryImageCreateInfo::handleTypes` when `image` was created

- If `memory` was created by a memory import operation, the external handle type of the imported memory **must** also have been set in `VkExternalMemoryImageCreateInfo` `::handleTypes` when `image` was created

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkBindImageMemoryDeviceGroupInfo`, `VkBindImageMemorySwapchainInfoKHR`, or `VkBindImagePlaneMemoryInfo`
- Each `sType` member in the `pNext` chain **must** be unique
- `image` **must** be a valid `VkImage` handle
- Both of `image`, and `memory` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

```
typedef struct VkBindImageMemoryDeviceGroupInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           deviceIndexCount;
    const uint32_t*    pDeviceIndices;
    uint32_t           splitInstanceBindRegionCount;
    const VkRect2D*    pSplitInstanceBindRegions;
} VkBindImageMemoryDeviceGroupInfo;
```

or the equivalent

```
typedef VkBindImageMemoryDeviceGroupInfo VkBindImageMemoryDeviceGroupInfoKHR;
```

If the `pNext` list of `VkBindImageMemoryInfo` includes a `VkBindImageMemoryDeviceGroupInfo` structure, then that structure determines how memory is bound to images across multiple devices in a device group.

The `VkBindImageMemoryDeviceGroupInfo` structure is defined as:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `deviceIndexCount` is the number of elements in `pDeviceIndices`.
- `pDeviceIndices` is a pointer to an array of device indices.
- `splitInstanceBindRegionCount` is the number of elements in `pSplitInstanceBindRegions`.
- `pSplitInstanceBindRegions` is a pointer to an array of `VkRect2D` structures describing which regions of the image are attached to each instance of memory.

If `deviceIndexCount` is greater than zero, then on device index `i` `image` is attached to the instance of the memory on the physical device with device index `pDeviceIndices[i]`.

Let  $N$  be the number of physical devices in the logical device. If `splitInstanceBindRegionCount` is greater than zero, then `pSplitInstanceBindRegions` is an array of  $N^2$  rectangles, where the image region specified by the rectangle at element  $i*N+j$  in resource instance  $i$  is bound to the memory instance  $j$ . The blocks of the memory that are bound to each sparse image block region use an offset in memory, relative to `memoryOffset`, computed as if the whole image were being bound to a contiguous range of memory. In other words, horizontally adjacent image blocks use consecutive blocks of memory, vertically adjacent image blocks are separated by the number of bytes per block multiplied by the width in blocks of `image`, and the block at  $(0,0)$  corresponds to memory starting at `memoryOffset`.

If `splitInstanceBindRegionCount` and `deviceIndexCount` are zero and the memory comes from a memory heap with the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains consecutive indices from zero to the number of physical devices in the logical device, minus one. In other words, by default each physical device attaches to its own instance of the memory.

If `splitInstanceBindRegionCount` and `deviceIndexCount` are zero and the memory comes from a memory heap without the `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` bit set, then it is as if `pDeviceIndices` contains an array of zeros. In other words, by default each physical device attaches to instance zero.

## Valid Usage

- At least one of `deviceIndexCount` and `splitInstanceBindRegionCount` **must** be zero.
- `deviceIndexCount` **must** either be zero or equal to the number of physical devices in the logical device
- All elements of `pDeviceIndices` **must** be valid device indices.
- `splitInstanceBindRegionCount` **must** either be zero or equal to the number of physical devices in the logical device squared
- Elements of `pSplitInstanceBindRegions` that correspond to the same instance of an image **must** not overlap.
- The `offset.x` member of any element of `pSplitInstanceBindRegions` **must** be a multiple of the sparse image block width (`VkSparseImageFormatProperties::imageGranularity.width`) of all non-metadata aspects of the image
- The `offset.y` member of any element of `pSplitInstanceBindRegions` **must** be a multiple of the sparse image block height (`VkSparseImageFormatProperties::imageGranularity.height`) of all non-metadata aspects of the image
- The `extent.width` member of any element of `pSplitInstanceBindRegions` **must** either be a multiple of the sparse image block width of all non-metadata aspects of the image, or else `extent.width + offset.x` **must** equal the width of the image subresource
- The `extent.height` member of any element of `pSplitInstanceBindRegions` **must** either be a multiple of the sparse image block height of all non-metadata aspects of the image, or else `extent.height + offset.y` **must** equal the width of the image subresource

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO`
- If `deviceIndexCount` is not `0`, `pDeviceIndices` **must** be a valid pointer to an array of `deviceIndexCount uint32_t` values
- If `splitInstanceBindRegionCount` is not `0`, `pSplitInstanceBindRegions` **must** be a valid pointer to an array of `splitInstanceBindRegionCount VkRect2D` structures

If the `pNext` chain of `VkBindImageMemoryInfo` includes a `VkBindImageMemorySwapchainInfoKHR` structure, then that structure includes a swapchain handle and image index indicating that the image will be bound to memory from that swapchain.

The `VkBindImageMemorySwapchainInfoKHR` structure is defined as:

```
typedef struct VkBindImageMemorySwapchainInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
    uint32_t           imageIndex;
} VkBindImageMemorySwapchainInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `swapchain` is `VK_NULL_HANDLE` or a swapchain handle.
- `imageIndex` is an image index within `swapchain`.

If `swapchain` is not `NULL`, the `swapchain` and `imageIndex` are used to determine the memory that the image is bound to, instead of `memory` and `memoryOffset`.

Memory **can** be bound to a swapchain and use the `pDeviceIndices` or `pSplitInstanceBindRegions` members of `VkBindImageMemoryDeviceGroupInfo`.

## Valid Usage

- `imageIndex` **must** be less than the number of images in `swapchain`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR`
- `swapchain` **must** be a valid `VkSwapchainKHR` handle

## Host Synchronization

- Host access to `swapchain` **must** be externally synchronized

In order to bind *planes* of a *disjoint image*, include a `VkBindImagePlaneMemoryInfo` structure in the `pNext` chain of `VkBindImageMemoryInfo`.

The `VkBindImagePlaneMemoryInfo` structure is defined as:

```
typedef struct VkBindImagePlaneMemoryInfo {
    VkStructureType      sType;
    const void*          pNext;
    VkImageAspectFlagBits planeAspect;
} VkBindImagePlaneMemoryInfo;
```

or the equivalent

```
typedef VkBindImagePlaneMemoryInfo VkBindImagePlaneMemoryInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `planeAspect` is the aspect of the disjoint image plane to bind.

## Valid Usage

- If the image's tiling is `VK_IMAGE_TILING_LINEAR` or `VK_IMAGE_TILING_OPTIMAL`, then `planeAspect` **must** be a single valid *format plane* for the image. (That is, `planeAspect` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT` for “`_2PLANE`” formats and `planeAspect` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT` for “`_3PLANE`” formats.)
- If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `planeAspect` **must** be a single valid *memory plane* for the image. (That is, `aspectMask` **must** specify a plane index that is less than the `drmFormatModifierPlaneCount` associated with the image's `format` and `drmFormatModifier`.)
- A single call to `vkBindImageMemory2` **must** bind all or none of the planes of an image (i.e. bindings to all planes of an image **must** be made in a single `vkBindImageMemory2` call), as separate bindings

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO`
- `planeAspect` **must** be a valid `VkImageAspectFlagBits` value

### Buffer-Image Granularity

There is an implementation-dependent limit, `bufferImageGranularity`, which specifies a page-like granularity at which linear and non-linear resources **must** be placed in adjacent memory locations to avoid aliasing. Two resources which do not satisfy this granularity requirement are said to `alias`. `bufferImageGranularity` is specified in bytes, and **must** be a power of two. Implementations which do not impose a granularity restriction **may** report a `bufferImageGranularity` value of one.

#### Note



Despite its name, `bufferImageGranularity` is really a granularity between “linear” and “non-linear” resources.

Given `resourceA` at the lower memory offset and `resourceB` at the higher memory offset in the same `VkDeviceMemory` object, where one resource is linear and the other is non-linear (as defined in the [Glossary](#)), and the following:

```
resourceA.end      = resourceA.memoryOffset + resourceA.size - 1
resourceA.endPage = resourceA.end & ~(bufferImageGranularity-1)
resourceB.start    = resourceB.memoryOffset
resourceB.startPage = resourceB.start & ~(bufferImageGranularity-1)
```

The following property **must** hold:

```
resourceA.endPage < resourceB.startPage
```

That is, the end of the first resource (A) and the beginning of the second resource (B) **must** be on separate “pages” of size `bufferImageGranularity`. `bufferImageGranularity` **may** be different than the physical page size of the memory heap. This restriction is only needed when a linear resource and a non-linear resource are adjacent in memory and will be used simultaneously. The memory ranges of adjacent resources **can** be closer than `bufferImageGranularity`, provided they meet the `alignment` requirement for the objects in question.

Sparse block size in bytes and sparse image and buffer memory alignments **must** all be multiples of the `bufferImageGranularity`. Therefore, memory bound to sparse resources naturally satisfies the `bufferImageGranularity`.

## 11.7. Resource Sharing Mode

Buffer and image objects are created with a *sharing mode* controlling how they **can** be accessed from queues. The supported sharing modes are:

```
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
    VK_SHARING_MODE_MAX_ENUM = 0x7FFFFFFF
} VkSharingMode;
```

- `VK_SHARING_MODE_EXCLUSIVE` specifies that access to any range or image subresource of the object will be exclusive to a single queue family at a time.
- `VK_SHARING_MODE_CONCURRENT` specifies that concurrent access to any range or image subresource of the object from multiple queue families is supported.

*Note*



`VK_SHARING_MODE_CONCURRENT` **may** result in lower performance access to the buffer or image than `VK_SHARING_MODE_EXCLUSIVE`.

Ranges of buffers and image subresources of image objects created using `VK_SHARING_MODE_EXCLUSIVE` **must** only be accessed by queues in the queue family that has *ownership* of the resource. Upon creation, such resources are not owned by any queue family; ownership is implicitly acquired upon first use within a queue. Once a resource using `VK_SHARING_MODE_EXCLUSIVE` is owned by some queue family, the application **must** perform a `queue family ownership transfer` to make the memory

contents of a range or image subresource accessible to a different queue family.

*Note*



Images still require a [layout transition](#) from `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED` before being used on the first queue.

A queue family **can** take ownership of an image subresource or buffer range of a resource created with `VK_SHARING_MODE_EXCLUSIVE`, without an ownership transfer, in the same way as for a resource that was just created; however, taking ownership in this way has the effect that the contents of the image subresource or buffer range are undefined.

Ranges of buffers and image subresources of image objects created using `VK_SHARING_MODE_CONCURRENT` **must** only be accessed by queues from the queue families specified through the `queueFamilyIndexCount` and `pQueueFamilyIndices` members of the corresponding create info structures.

### 11.7.1. External Resource Sharing

Resources **should** only be accessed in the Vulkan instance that has exclusive ownership of their underlying memory. Only one Vulkan instance has exclusive ownership of a resource's underlying memory at a given time, regardless of whether the resource was created using `VK_SHARING_MODE_EXCLUSIVE` or `VK_SHARING_MODE_CONCURRENT`. Applications can transfer ownership of a resource's underlying memory only if the memory has been imported from or exported to another instance or external API using external memory handles. The semantics for transferring ownership outside of the instance are similar to those used for transferring ownership of `VK_SHARING_MODE_EXCLUSIVE` resources between queues, and is also accomplished using `VkBufferMemoryBarrier` or `VkImageMemoryBarrier` operations. Applications **must**

1. Release exclusive ownership from the source instance or API.
2. Ensure the release operation has completed using semaphores or fences.
3. Acquire exclusive ownership in the destination instance or API

Unlike queue ownership transfers, the destination instance or API is not specified explicitly when releasing ownership, nor is the source instance or API specified when acquiring ownership. Instead, the image or memory barrier's `dstQueueFamilyIndex` or `srcQueueFamilyIndex` parameters are set to the reserved queue family index `VK_QUEUE_FAMILY_EXTERNAL` or `VK_QUEUE_FAMILY_FOREIGN_EXT` to represent the external destination or source respectively.

Binding a resource to a memory object shared between multiple Vulkan instances or other APIs does not change the ownership of the underlying memory. The first entity to access the resource implicitly acquires ownership. Accessing a resource backed by memory that is owned by a particular instance or API has the same semantics as accessing a `VK_SHARING_MODE_EXCLUSIVE` resource, with one exception: Implementations **must** ensure layout transitions performed on one member of a set of identical subresources of identical images that alias the same range of an underlying memory object affect the layout of all the subresources in the set.

As a corollary, writes to any image subresources in such a set **must** not make the contents of memory used by other subresources in the set undefined. An application **can** define the content of

a subresource of one image by performing device writes to an identical subresource of another image provided both images are bound to the same region of external memory. Applications **may** also add resources to such a set after the content of the existing set members has been defined without making the content undefined by creating a new image with the initial layout `VK_IMAGE_LAYOUT_UNDEFINED` and binding it to the same region of external memory as the existing images.

*Note*

Because layout transitions apply to all identical images aliasing the same region of external memory, the actual layout of the memory backing a new image as well as an existing image with defined content will not be undefined. Such an image is not usable until it acquires ownership of its memory from the existing owner. Therefore, the layout specified as part of this transition will be the true initial layout of the image. The undefined layout specified when creating it is a placeholder to simplify valid usage requirements.



## 11.8. Memory Aliasing

A range of a `VkDeviceMemory` allocation is *aliased* if it is bound to multiple resources simultaneously, as described below, via `vkBindImageMemory`, `vkBindBufferMemory`, via [sparse memory bindings](#), or by binding the memory to resources in multiple Vulkan instances or external APIs using external memory handle export and import mechanisms.

Consider two resources,  $\text{resource}_A$  and  $\text{resource}_B$ , bound respectively to memory range<sub>A</sub> and range<sub>B</sub>. Let paddedRange<sub>A</sub> and paddedRange<sub>B</sub> be, respectively, range<sub>A</sub> and range<sub>B</sub> aligned to [bufferImageGranularity](#). If the resources are both linear or both non-linear (as defined in the [Glossary](#)), then the resources *alias* the memory in the intersection of range<sub>A</sub> and range<sub>B</sub>. If one resource is linear and the other is non-linear, then the resources *alias* the memory in the intersection of paddedRange<sub>A</sub> and paddedRange<sub>B</sub>.

Applications **can** alias memory, but use of multiple aliases is subject to several constraints.

*Note*



Memory aliasing **can** be useful to reduce the total device memory footprint of an application, if some large resources are used for disjoint periods of time.

When a [non-linear](#), non-`VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` image is bound to an aliased range, all image subresources of the image *overlap* the range. When a linear image is bound to an aliased range, the image subresources that (according to the image's advertised layout) include bytes from the aliased range overlap the range. When a `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` image has sparse image blocks bound to an aliased range, only image subresources including those sparse image blocks overlap the range, and when the memory bound to the image's mip tail overlaps an aliased range all image subresources in the mip tail overlap the range.

Buffers, and linear image subresources in either the `VK_IMAGE_LAYOUT_PREINITIALIZED` or `VK_IMAGE_LAYOUT_GENERAL` layouts, are *host-accessible subresources*. That is, the host has a well-defined addressing scheme to interpret the contents, and thus the layout of the data in memory **can**

be consistently interpreted across aliases if each of those aliases is a host-accessible subresource. Non-linear images, and linear image subresources in other layouts, are not host-accessible.

If two aliases are both host-accessible, then they interpret the contents of the memory in consistent ways, and data written to one alias **can** be read by the other alias.

If two aliases are both images that were created with identical creation parameters, both were created with the `VK_IMAGE_CREATE_ALIAS_BIT` flag set, and both are bound identically to memory except for `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` and `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions`, then they interpret the contents of the memory in consistent ways, and data written to one alias **can** be read by the other alias.

Additionally, if an individual plane of a multi-planar image and a single-plane image alias the same memory, then they also interpret the contents of the memory in consistent ways under the same conditions, but with the following modifications:

- Both **must** have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` flag.
- The single-plane image **must** have a `VkFormat` that is **equivalent** to that of the multi-planar image's individual plane.
- The single-plane image and the individual plane of the multi-planar image **must** be bound identically to memory except for `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` and `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions`.
- The `width` and `height` of the single-plane image are derived from the multi-planar image's dimensions in the manner listed for `plane compatibility` for the aliased plane.
- If either image's `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then both images **must** be `linear`.
- All other creation parameters **must** be identical

Aliases created by binding the same memory to resources in multiple Vulkan instances or external APIs using external memory handle export and import mechanisms interpret the contents of the memory in consistent ways, and data written to one alias **can** be read by the other alias.

Otherwise, the aliases interpret the contents of the memory differently, and writes via one alias make the contents of memory partially or completely undefined to the other alias. If the first alias is a host-accessible subresource, then the bytes affected are those written by the memory operations according to its addressing scheme. If the first alias is not host-accessible, then the bytes affected are those overlapped by the image subresources that were written. If the second alias is a host-accessible subresource, the affected bytes become undefined. If the second alias is not host-accessible, all sparse image blocks (for sparse partially-resident images) or all image subresources (for non-sparse image and fully resident sparse images) that overlap the affected bytes become undefined.

If any image subresources are made undefined due to writes to an alias, then each of those image subresources **must** have its layout transitioned from `VK_IMAGE_LAYOUT_UNDEFINED` to a valid layout before it is used, or from `VK_IMAGE_LAYOUT_PREINITIALIZED` if the memory has been written by the host. If any sparse blocks of a sparse image have been made undefined, then only the image subresources containing them **must** be transitioned.

Use of an overlapping range by two aliases **must** be separated by a memory dependency using the appropriate [access types](#) if at least one of those uses performs writes, whether the aliases interpret memory consistently or not. If buffer or image memory barriers are used, the scope of the barrier **must** contain the entire range and/or set of image subresources that overlap.

If two aliasing image views are used in the same framebuffer, then the render pass **must** declare the attachments using the [VK\\_ATTACHMENT\\_DESCRIPTION\\_MAY\\_ALIAS\\_BIT](#), and follow the other rules listed in that section.

*Note*



Memory recycled via an application suballocator (i.e. without freeing and reallocating the memory objects) is not substantially different from memory aliasing. However, a suballocator usually waits on a fence before recycling a region of memory, and signaling a fence involves sufficient implicit dependencies to satisfy all the above requirements.

## 11.9. Acceleration Structures

Acceleration structures are an opaque structure that is built by the implementation to more efficiently perform spatial queries on the provided geometric data. For this extension, an acceleration structure is either a top-level acceleration structure containing a set of bottom-level acceleration structures or a bottom-level acceleration structure containing either a set of axis-aligned bounding boxes for custom geometry or a set of triangles.

Each instance in the top-level acceleration structure contains a reference to a bottom-level acceleration structure as well as an instance transform plus information required to index into the shader bindings. The top-level acceleration structure is what is bound to the acceleration descriptor to trace inside the shader in the ray tracing pipeline.

Acceleration structures are represented by [VkAccelerationStructureNV](#) handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkAccelerationStructureNV)
```

To create acceleration structures, call:

```
VkResult vkCreateAccelerationStructureNV(  
    VkDevice                                     device,  
    const VkAccelerationStructureCreateInfoNV* pCreateInfo,  
    const VkAllocationCallbacks*                 pAllocator,  
    VkAccelerationStructureNV*                  pAccelerationStructure);
```

- **device** is the logical device that creates the buffer object.
- **pCreateInfo** is a pointer to a [VkAccelerationStructureCreateInfoNV](#) structure containing parameters affecting creation of the acceleration structure.
- **pAllocator** controls host memory allocation as described in the [Memory Allocation](#) chapter.

- `pAccelerationStructure` is a pointer to a `VkAccelerationStructureNV` handle in which the resulting acceleration structure object is returned.

Similar to other objects in Vulkan, the acceleration structure creation merely creates an object with a specific “shape” as specified by the information in `VkAccelerationStructureCreateInfoNV` and `compactedSize` in `pCreateInfo`. Populating the data in the object after allocating and binding memory is done with `vkCmdBuildAccelerationStructureNV` and `vkCmdCopyAccelerationStructureNV`.

Acceleration structure creation uses the count and type information from the geometries, but does not use the data references in the structures.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkAccelerationStructureCreateInfoNV` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pAccelerationStructure` **must** be a valid pointer to a `VkAccelerationStructureNV` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkAccelerationStructureCreateInfoNV` structure is defined as:

```
typedef struct VkAccelerationStructureCreateInfoNV {
    VkStructureType           sType;
    const void*               pNext;
    VkDeviceSize              compactedSize;
    VkAccelerationStructureInfoNV info;
} VkAccelerationStructureCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `compactedSize` is the size from the result of `vkCmdWriteAccelerationStructuresPropertiesNV` if this acceleration structure is going to be the target of a compacting copy.
- `info` is the `VkAccelerationStructureInfoNV` structure specifying further parameters of the created acceleration structure.

## Valid Usage

- If `compactedSize` is not `0` then both `info.geometryCount` and `info.instanceCount` **must** be `0`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_NV`
- `pNext` **must** be `NULL`
- `info` **must** be a valid `VkAccelerationStructureCreateInfoNV` structure

The `VkAccelerationStructureCreateInfoNV` structure is defined as:

```
typedef struct VkAccelerationStructureCreateInfoNV {  
    VkStructureType sType;  
    const void* pNext;  
    VkAccelerationStructureTypeNV type;  
    VkBuildAccelerationStructureFlagsNV flags;  
    uint32_t instanceCount;  
    uint32_t geometryCount;  
    const VkGeometryNV* pGeometries;  
} VkAccelerationStructureCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `type` is a `VkAccelerationStructureTypeNV` value specifying the type of acceleration structure that will be created.
- `flags` is a bitmask of `VkBuildAccelerationStructureFlagBitsNV` specifying additional parameters of the acceleration structure.
- `instanceCount` specifies the number of instances that will be in the new acceleration structure.
- `geometryCount` specifies the number of geometries that will be in the new acceleration structure.
- `pGeometries` is a pointer to an array of `geometryCount` `VkGeometryNV` structures containing the scene data being passed into the acceleration structure.

`VkAccelerationStructureCreateInfoNV` contains information that is used both for acceleration structure creation with `vkCreateAccelerationStructureNV` and in combination with the actual geometric data to build the acceleration structure with `vkCmdBuildAccelerationStructureNV`.

## Valid Usage

- `geometryCount` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxGeometryCount`
- `instanceCount` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxInstanceCount`
- The total number of triangles in all geometries **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxTriangleCount`
- If `type` is `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_NV` then `geometryCount` **must** be `0`
- If `type` is `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV` then `instanceCount` **must** be `0`
- If `type` is `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV` then the `geometryType` member of each geometry in `pGeometries` **must** be the same
- If `flags` has the `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV` bit set, then it **must** not have the `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_NV` bit set
- `scratch` **must** have been created with `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` usage flag
- If `instanceData` is not `VK_NULL_HANDLE`, `instanceData` **must** have been created with `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` usage flag

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_INFO_NV`
- `pNext` **must** be `NULL`
- `type` **must** be a valid `VkAccelerationStructureTypeNV` value
- `flags` **must** be a valid combination of `VkBuildAccelerationStructureFlagBitsNV` values
- If `geometryCount` is not `0`, `pGeometries` **must** be a valid pointer to an array of `geometryCount` valid `VkGeometryNV` structures

Values which **can** be set in `VkAccelerationStructureInfoNV::type`, specifying the type of acceleration structure, are:

```
typedef enum VkAccelerationStructureTypeNV {
    VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_NV = 0,
    VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV = 1,
    VK_ACCELERATION_STRUCTURE_TYPE_MAX_ENUM_NV = 0x7FFFFFFF
} VkAccelerationStructureTypeNV;
```

- `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_NV` is a top-level acceleration structure containing instance data referring to bottom-level level acceleration structures.
- `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV` is a bottom-level acceleration structure containing the AABBs or geometry to be intersected.

Bits which **can** be set in `VkAccelerationStructureInfoNV::flags`, specifying additional parameters for acceleration structure builds, are:

```
typedef enum VkBuildAccelerationStructureFlagBitsNV {
    VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV = 0x00000001,
    VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_NV = 0x00000002,
    VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV = 0x00000004,
    VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_NV = 0x00000008,
    VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_NV = 0x00000010,
    VK_BUILD_ACCELERATION_STRUCTURE_FLAG_BITS_MAX_ENUM_NV = 0x7FFFFFFF
} VkBuildAccelerationStructureFlagBitsNV;
```

- `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV` indicates that the specified acceleration structure **can** be updated with `update` of `VK_TRUE` in `vkCmdBuildAccelerationStructureNV`.
- `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_NV` indicates that the specified acceleration structure **can** act as the source for `vkCmdCopyAccelerationStructureNV` with `mode` of `VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_NV` to produce a compacted acceleration structure.
- `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV` indicates that the given acceleration structure build **should** prioritize trace performance over build time.
- `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_NV` indicates that the given acceleration structure build **should** prioritize build time over trace performance.
- `VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_NV` indicates that this acceleration structure **should** minimize the size of the scratch memory and the final result build, potentially at the expense of build time or trace performance.

*Note*



`VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV` and `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_NV` **may** take more time and memory than a normal build, and so **should** only be used when those features are used.

```
typedef VkFlags VkBuildAccelerationStructureFlagsNV;
```

`VkBuildAccelerationStructureFlagsNV` is a bitmask type for setting a mask of zero or more `VkBuildAccelerationStructureFlagBitsNV`.

The `VkGeometryNV` structure is defined as:

```

typedef struct VkGeometryNV {
    VkStructureType      sType;
    const void*        pNext;
    VkGeometryTypeNV    geometryType;
    VkGeometryDataNV    geometry;
    VkGeometryFlagsNV   flags;
} VkGeometryNV;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **geometryType** describes which type of geometry this **VkGeometryNV** refers to.
- **geometry** contains the geometry data as described in **VkGeometryDataNV**.
- **flags** has flags describing options for this geometry.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_GEOMETRY\_NV**
- **pNext** **must** be **NULL**
- **geometryType** **must** be a valid **VkGeometryTypeNV** value
- **geometry** **must** be a valid **VkGeometryDataNV** structure
- **flags** **must** be a valid combination of **VkGeometryFlagBitsNV** values

Geometry types are specified by **VkGeometryTypeNV**, which takes values:

```

typedef enum VkGeometryTypeNV {
    VK_GEOMETRY_TYPE_TRIANGLES_NV = 0,
    VK_GEOMETRY_TYPE_AABBS_NV = 1,
    VK_GEOMETRY_TYPE_MAX_ENUM_NV = 0x7FFFFFFF
} VkGeometryTypeNV;

```

- **VK\_GEOMETRY\_TYPE\_TRIANGLES\_NV** indicates that the **triangles** of **VkGeometryDataNV** contains valid data.
- **VK\_GEOMETRY\_TYPE\_AABBS\_NV** indicates that the **aabbs** of **VkGeometryDataNV** contains valid data.

Bits which **can** be set in **VkGeometryNV::flags**, specifying additional parameters for acceleration structure builds, are:

```
typedef enum VkGeometryFlagBitsNV {
    VK_GEOMETRY_OPAQUE_BIT_NV = 0x00000001,
    VK_GEOMETRY_NO_DUPLICATE_ANY_HIT_INVOCATION_BIT_NV = 0x00000002,
    VK_GEOMETRY_FLAG_BITS_MAX_ENUM_NV = 0x7FFFFFFF
} VkGeometryFlagBitsNV;
```

- `VK_GEOMETRY_OPAQUE_BIT_NV` indicates that this geometry does not invoke the any-hit shaders even if present in a hit group.
- `VK_GEOMETRY_NO_DUPLICATE_ANY_HIT_INVOCATION_BIT_NV` indicates that the implementation **must** only call the any-hit shader a single time for each primitive in this geometry. If this bit is absent an implementation **may** invoke the any-hit shader more than once for this geometry.

```
typedef VkFlags VkGeometryFlagsNV;
```

`VkGeometryFlagsNV` is a bitmask type for setting a mask of zero or more `VkGeometryFlagBitsNV`.

The `VkGeometryDataNV` structure is defined as:

```
typedef struct VkGeometryDataNV {
    VkGeometryTrianglesNV    triangles;
    VkGeometryAABBNV        aabbs;
} VkGeometryDataNV;
```

- `triangles` contains triangle data if `VkGeometryNV::geometryType` is `VK_GEOMETRY_TYPE_TRIANGLES_NV`.
- `aabbs` contains axis-aligned bounding box data if `VkGeometryNV::geometryType` is `VK_GEOMETRY_TYPE_AABBS_NV`.

### Valid Usage (Implicit)

- `triangles` **must** be a valid `VkGeometryTrianglesNV` structure
- `aabbs` **must** be a valid `VkGeometryAABBNV` structure

The `VkGeometryTrianglesNV` structure is defined as:

```

typedef struct VkGeometryTrianglesNV {
    VkStructureType      sType;
    const void*        pNext;
    VkBuffer            vertexData;
    VkDeviceSize         vertexOffset;
    uint32_t             vertexCount;
    VkDeviceSize         vertexStride;
    VkFormat             vertexFormat;
    VkBuffer            indexData;
    VkDeviceSize         indexOffset;
    uint32_t             indexCount;
    VkIndexType          indexType;
    VkBuffer            transformData;
    VkDeviceSize         transformOffset;
} VkGeometryTrianglesNV;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **vertexData** is the buffer containing vertex data for this geometry.
- **vertexOffset** is the offset in bytes within **vertexData** containing vertex data for this geometry.
- **vertexCount** is the number of valid vertices.
- **vertexStride** is the stride in bytes between each vertex.
- **vertexFormat** is the format of each vertex element.
- **indexData** is the buffer containing index data for this geometry.
- **indexOffset** is the offset in bytes within **indexData** containing index data for this geometry.
- **indexCount** is the number of indices to include in this geometry.
- **indexType** is the format of each index.
- **transformData** is a buffer containing optional reference to an array of 32-bit floats representing a 3x4 row major affine transformation matrix for this geometry.
- **transformOffset** is the offset in bytes in **transformData** of the transform information described above.

If **indexType** is **VK\_INDEX\_TYPE\_NONE\_NV**, then this structure describes a set of triangles determined by **vertexCount**. Otherwise, this structure describes a set of indexed triangles determined by **indexCount**.

## Valid Usage

- `vertexOffset` **must** be less than the size of `vertexData`
- `vertexOffset` **must** be a multiple of the component size of `vertexFormat`
- `vertexFormat` **must** be one of `VK_FORMAT_R32G32B32_SFLOAT`, `VK_FORMAT_R32G32_SFLOAT`, `VK_FORMAT_R16G16B16_SFLOAT`, `VK_FORMAT_R16G16_SFLOAT`, `VK_FORMAT_R16G16_SNORM`, or `VK_FORMAT_R16G16B16_SNORM`
- `indexOffset` **must** be less than the size of `indexData`
- `indexOffset` **must** be a multiple of the element size of `indexType`
- `indexType` **must** be `VK_INDEX_TYPE_UINT16`, `VK_INDEX_TYPE_UINT32`, or `VK_INDEX_TYPE_NONE_NV`
- `indexData` **must** be `VK_NULL_HANDLE` if `indexType` is `VK_INDEX_TYPE_NONE_NV`
- `indexData` **must** be a valid `VkBuffer` handle if `indexType` is not `VK_INDEX_TYPE_NONE_NV`
- `indexCount` **must** be `0` if `indexType` is `VK_INDEX_TYPE_NONE_NV`
- `transformOffset` **must** be less than the size of `transformData`
- `transformOffset` **must** be a multiple of `16`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_GEOMETRY_TRIANGLES_NV`
- `pNext` **must** be `NULL`
- If `vertexData` is not `VK_NULL_HANDLE`, `vertexData` **must** be a valid `VkBuffer` handle
- `vertexFormat` **must** be a valid `VkFormat` value
- If `indexData` is not `VK_NULL_HANDLE`, `indexData` **must** be a valid `VkBuffer` handle
- `indexType` **must** be a valid `VkIndexType` value
- If `transformData` is not `VK_NULL_HANDLE`, `transformData` **must** be a valid `VkBuffer` handle
- Each of `indexData`, `transformData`, and `vertexData` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

The `VkGeometryAABBNV` structure is defined as:

```
typedef struct VkGeometryAABBNV {
    VkStructureType    sType;
    const void*        pNext;
    VkBuffer           aabbData;
    uint32_t           numAABBS;
    uint32_t           stride;
    VkDeviceSize        offset;
} VkGeometryAABBNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `aabbData` is the buffer containing axis-aligned bounding box data.
- `numAABBs` is the number of AABBs in this geometry.
- `stride` is the stride in bytes between AABBs in `aabbData`.
- `offset` is the offset in bytes of the first AABB in `aabbData`.

The AABB data in memory is six 32-bit floats consisting of the minimum x, y, and z values followed by the maximum x, y, and z values.

## Valid Usage

- `offset` **must** be less than the size of `aabbData`
- `offset` **must** be a multiple of 8
- `stride` **must** be a multiple of 8

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_GEOMETRY_AABB_NV`
- `pNext` **must** be `NULL`
- If `aabbData` is not `VK_NULL_HANDLE`, `aabbData` **must** be a valid `VkBuffer` handle

To destroy an acceleration structure, call:

```
void vkDestroyAccelerationStructureNV(
    VkDevice                                     device,
    VkAccelerationStructureNV                    accelerationStructure,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the buffer.
- `accelerationStructure` is the acceleration structure to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `accelerationStructure` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `accelerationStructure` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `accelerationStructure` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `accelerationStructure` **must** be a valid `VkAccelerationStructureNV` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `accelerationStructure` **must** have been created, allocated, or retrieved from `device`

An acceleration structure has memory requirements for the structure object itself, scratch space for the build, and scratch space for the update.

To query the memory requirements call:

```
void vkGetAccelerationStructureMemoryRequirementsNV(  
    VkDevice                               device,  
    const VkAccelerationStructureMemoryRequirementsInfoNV* pInfo,  
    VkMemoryRequirements2KHR*               pMemoryRequirements);
```

- `device` is the logical device on which the acceleration structure was created.
- `pInfo` specifies the acceleration structure to get memory requirements for.
- `pMemoryRequirements` returns the requested acceleration structure memory requirements.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkAccelerationStructureMemoryRequirementsInfoNV` structure
- `pMemoryRequirements` **must** be a valid pointer to a `VkMemoryRequirements2KHR` structure

The `VkAccelerationStructureMemoryRequirementsInfoNV` structure is defined as:

```

typedef struct VkAccelerationStructureMemoryRequirementsInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkAccelerationStructureMemoryRequirementsTypeNV type;
    VkAccelerationStructureNV accelerationStructure;
} VkAccelerationStructureMemoryRequirementsInfoNV;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **type** selects the type of memory requirement being queried.
  - VK\_ACCELERATION\_STRUCTURE\_MEMORY\_REQUIREMENTS\_TYPE\_OBJECT\_NV** returns the memory requirements for the object itself.
  - VK\_ACCELERATION\_STRUCTURE\_MEMORY\_REQUIREMENTS\_TYPE\_BUILD\_SCRATCH\_NV** returns the memory requirements for the scratch memory when doing a build.
  - VK\_ACCELERATION\_STRUCTURE\_MEMORY\_REQUIREMENTS\_TYPE\_UPDATE\_SCRATCH\_NV** returns the memory requirements for the scratch memory when doing an update.
- **accelerationStructure** is the acceleration structure to be queried for memory requirements.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_ACCELERATION\_STRUCTURE\_MEMORY\_REQUIREMENTS\_INFO\_NV**
- **pNext** **must** be **NULL**
- **type** **must** be a valid **VkAccelerationStructureMemoryRequirementsTypeNV** value
- **accelerationStructure** **must** be a valid **VkAccelerationStructureNV** handle

Possible values of **type** in **VkAccelerationStructureMemoryRequirementsInfoNV** are:

```

typedef enum VkAccelerationStructureMemoryRequirementsTypeNV {
    VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV = 0,
    VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_BUILD_SCRATCH_NV = 1,
    VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_UPDATE_SCRATCH_NV = 2,
    VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_MAX_ENUM_NV = 0x7FFFFFFF
} VkAccelerationStructureMemoryRequirementsTypeNV;

```

- **VK\_ACCELERATION\_STRUCTURE\_MEMORY\_REQUIREMENTS\_TYPE\_OBJECT\_NV** requests the memory requirement for the **VkAccelerationStructureNV** backing store.
- **VK\_ACCELERATION\_STRUCTURE\_MEMORY\_REQUIREMENTS\_TYPE\_BUILD\_SCRATCH\_NV** requests the memory requirement for scratch space during the initial build.
- **VK\_ACCELERATION\_STRUCTURE\_MEMORY\_REQUIREMENTS\_TYPE\_UPDATE\_SCRATCH\_NV** requests the memory requirement for scratch space during an update.

To attach memory to one or more acceleration structures at a time, call:

```
VkResult vkBindAccelerationStructureMemoryNV(  
    VkDevice device,  
    uint32_t bindInfoCount,  
    const VkBindAccelerationStructureMemoryInfoNV* pBindInfos);
```

- `device` is the logical device that owns the acceleration structures and memory.
- `bindInfoCount` is the number of elements in `pBindInfos`.
- `pBindInfos` is a pointer to an array of `VkBindAccelerationStructureMemoryInfoNV` structures describing images and memory to bind.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pBindInfos` **must** be a valid pointer to an array of `bindInfoCount` valid `VkBindAccelerationStructureMemoryInfoNV` structures
- `bindInfoCount` **must** be greater than 0

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkBindAccelerationStructureMemoryInfoNV` structure is defined as:

```
typedef struct VkBindAccelerationStructureMemoryInfoNV {  
    VkStructureType sType;  
    const void* pNext;  
    VkAccelerationStructureNV accelerationStructure;  
    VkDeviceMemory memory;  
    VkDeviceSize memoryOffset;  
    uint32_t deviceIndexCount;  
    const uint32_t* pDeviceIndices;  
} VkBindAccelerationStructureMemoryInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `accelerationStructure` is the acceleration structure to be attached to memory.
- `memory` is a `VkDeviceMemory` object describing the device memory to attach.

- `memoryOffset` is the start offset of the region of memory that is to be bound to the acceleration structure. The number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified acceleration structure.
- `deviceIndexCount` is the number of elements in `pDeviceIndices`.
- `pDeviceIndices` is a pointer to an array of device indices.

## Valid Usage

- `accelerationStructure` **must** not already be backed by a memory object
- `memoryOffset` **must** be less than the size of `memory`
- `memory` **must** have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetAccelerationStructureMemoryRequirementsNV` with `accelerationStructure` and `type` of `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV`
- `memoryOffset` **must** be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetAccelerationStructureMemoryRequirementsNV` with `accelerationStructure` and `type` of `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV`
- The `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetAccelerationStructureMemoryRequirementsNV` with `accelerationStructure` and `type` of `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV` **must** be less than or equal to the size of `memory` minus `memoryOffset`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BIND_ACCELERATION_STRUCTURE_MEMORY_INFO_NV`
- `pNext` **must** be `NULL`
- `accelerationStructure` **must** be a valid `VkAccelerationStructureNV` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- If `deviceIndexCount` is not `0`, `pDeviceIndices` **must** be a valid pointer to an array of `deviceIndexCount` `uint32_t` values
- Both of `accelerationStructure`, and `memory` **must** have been created, allocated, or retrieved from the same `VkDevice`

To allow constructing geometry instances with device code if desired, we need to be able to query a opaque handle for an acceleration structure. This handle is a value of 8 bytes. To get this handle, call:

```
VkResult vkGetAccelerationStructureHandleNV(  
    VkDevice device,  
    VkAccelerationStructureNV accelerationStructure,  
    size_t dataSize,  
    void* pData);
```

- `device` is the logical device that owns the acceleration structures.
- `accelerationStructure` is the acceleration structure.
- `dataSize` is the size in bytes of the buffer pointed to by `pData`.
- `pData` is a pointer to a user-allocated buffer where the results will be written.

## Valid Usage

- `dataSize` **must** be large enough to contain the result of the query, as described above
- `accelerationStructure` **must** be bound completely and contiguously to a single `VkDeviceMemory` object via `vkBindAccelerationStructureMemoryNV`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `accelerationStructure` **must** be a valid `VkAccelerationStructureNV` handle
- `pData` **must** be a valid pointer to an array of `dataSize` bytes
- `dataSize` **must** be greater than `0`
- `accelerationStructure` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

# Chapter 12. Samplers

`VkSampler` objects represent the state of an image sampler which is used by the implementation to read image data and apply filtering and other transformations for the shader.

Samplers are represented by `VkSampler` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSampler)
```

To create a sampler object, call:

```
VkResult vkCreateSampler(  
    VkDevice device,  
    const VkSamplerCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSampler* pSampler);
```

- `device` is the logical device that creates the sampler.
- `pCreateInfo` is a pointer to a `VkSamplerCreateInfo` structure specifying the state of the sampler object.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pSampler` is a pointer to a `VkSampler` handle in which the resulting sampler object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkSamplerCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSampler` **must** be a valid pointer to a `VkSampler` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_TOO_MANY_OBJECTS`

The `VkSamplerCreateInfo` structure is defined as:

```

typedef struct VkSamplerCreateInfo {
    VkStructureType      sType;
    const void*        pNext;
    VkSamplerCreateFlags flags;
    VkFilter             magFilter;
    VkFilter             minFilter;
    VkSamplerMipmapMode mipmapMode;
    VkSamplerAddressMode addressModeU;
    VkSamplerAddressMode addressModeV;
    VkSamplerAddressMode addressModeW;
    float              mipLodBias;
    VkBool32             anisotropyEnable;
    float              maxAnisotropy;
    VkBool32             compareEnable;
    VkCompareOp          compareOp;
    float              minLod;
    float              maxLod;
    VkBorderColor         borderColor;
    VkBool32             unnormalizedCoordinates;
} VkSamplerCreateInfo;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is a bitmask of **VkSamplerCreateFlagBits** describing additional parameters of the sampler.
- **magFilter** is a **VkFilter** value specifying the magnification filter to apply to lookups.
- **minFilter** is a **VkFilter** value specifying the minification filter to apply to lookups.
- **mipmapMode** is a **VkSamplerMipmapMode** value specifying the mipmap filter to apply to lookups.
- **addressModeU** is a **VkSamplerAddressMode** value specifying the addressing mode for outside [0..1] range for U coordinate.
- **addressModeV** is a **VkSamplerAddressMode** value specifying the addressing mode for outside [0..1] range for V coordinate.
- **addressModeW** is a **VkSamplerAddressMode** value specifying the addressing mode for outside [0..1] range for W coordinate.
- **mipLodBias** is the bias to be added to mipmap LOD (level-of-detail) calculation and bias provided by image sampling functions in SPIR-V, as described in the [Level-of-Detail Operation](#) section.
- **anisotropyEnable** is **VK\_TRUE** to enable anisotropic filtering, as described in the [Texel Anisotropic Filtering](#) section, or **VK\_FALSE** otherwise.
- **maxAnisotropy** is the anisotropy value clamp used by the sampler when **anisotropyEnable** is **VK\_TRUE**. If **anisotropyEnable** is **VK\_FALSE**, **maxAnisotropy** is ignored.
- **compareEnable** is **VK\_TRUE** to enable comparison against a reference value during lookups, or **VK\_FALSE** otherwise.
  - Note: Some implementations will default to shader state if this member does not match.

- `compareOp` is a [VkCompareOp](#) value specifying the comparison function to apply to fetched data before filtering as described in the [Depth Compare Operation](#) section.
- `minLod` and `maxLod` are the values used to clamp the computed LOD value, as described in the [Level-of-Detail Operation](#) section.
- `borderColor` is a [VkBorderColor](#) value specifying the predefined border color to use.
- `unnormalizedCoordinates` controls whether to use unnormalized or normalized texel coordinates to address texels of the image. When set to `VK_TRUE`, the range of the image coordinates used to lookup the texel is in the range of zero to the image dimensions for x, y and z. When set to `VK_FALSE` the range of image coordinates is zero to one.

When `unnormalizedCoordinates` is `VK_TRUE`, images the sampler is used with in the shader have the following requirements:

- The `viewType` **must** be either `VK_IMAGE_VIEW_TYPE_1D` or `VK_IMAGE_VIEW_TYPE_2D`.
- The image view **must** have a single layer and a single mip level.

When `unnormalizedCoordinates` is `VK_TRUE`, image built-in functions in the shader that use the sampler have the following requirements:

- The functions **must** not use projection.
- The functions **must** not use offsets.

#### *Mapping of OpenGL to Vulkan filter modes*

`magFilter` values of `VK_FILTER_NEAREST` and `VK_FILTER_LINEAR` directly correspond to `GL_NEAREST` and `GL_LINEAR` magnification filters. `minFilter` and `mipmapMode` combine to correspond to the similarly named OpenGL minification filter of `GL_minFilter_MIPMAP_mipmapMode` (e.g. `minFilter` of `VK_FILTER_LINEAR` and `mipmapMode` of `VK_SAMPLER_MIPMAP_MODE_NEAREST` correspond to `GL_LINEAR_MIPMAP_NEAREST`).



There are no Vulkan filter modes that directly correspond to OpenGL minification filters of `GL_LINEAR` or `GL_NEAREST`, but they **can** be emulated using `VK_SAMPLER_MIPMAP_MODE_NEAREST`, `minLod` = 0, and `maxLod` = 0.25, and using `minFilter` = `VK_FILTER_LINEAR` or `minFilter` = `VK_FILTER_NEAREST`, respectively.

Note that using a `maxLod` of zero would cause [magnification](#) to always be performed, and the `magFilter` to always be used. This is valid, just not an exact match for OpenGL behavior. Clamping the maximum LOD to 0.25 allows the  $\lambda$  value to be non-zero and minification to be performed, while still always rounding down to the base level. If the `minFilter` and `magFilter` are equal, then using a `maxLod` of zero also works.

The maximum number of sampler objects which **can** be simultaneously created on a device is implementation-dependent and specified by the `maxSamplerAllocationCount` member of the [VkPhysicalDeviceLimits](#) structure. If `maxSamplerAllocationCount` is exceeded, `vkCreateSampler` will return `VK_ERROR_TOO_MANY_OBJECTS`.

Since [VkSampler](#) is a non-dispatchable handle type, implementations **may** return the same handle

for sampler state vectors that are identical. In such cases, all such objects would only count once against the `maxSamplerAllocationCount` limit.

## Valid Usage

- The absolute value of `mipLodBias` **must** be less than or equal to `VkPhysicalDeviceLimits::maxSamplerLodBias`
- `maxLod` **must** be greater than or equal to `minLod`
- If the `anisotropic sampling` feature is not enabled, `anisotropyEnable` **must** be `VK_FALSE`
- If `anisotropyEnable` is `VK_TRUE`, `maxAnisotropy` **must** be between `1.0` and `VkPhysicalDeviceLimits::maxSamplerAnisotropy`, inclusive
- If `sampler Y'CBCR conversion` is enabled and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT` is not set for the format, `minFilter` and `magFilter` **must** be equal to the sampler `Y'CBCR conversion`'s `chromaFilter`
- If `unnormalizedCoordinates` is `VK_TRUE`, `minFilter` and `magFilter` **must** be equal
- If `unnormalizedCoordinates` is `VK_TRUE`, `mipmapMode` **must** be `VK_SAMPLER_MIPMAP_MODE_NEAREST`
- If `unnormalizedCoordinates` is `VK_TRUE`, `minLod` and `maxLod` **must** be zero
- If `unnormalizedCoordinates` is `VK_TRUE`, `addressModeU` and `addressModeV` **must** each be either `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` or `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`
- If `unnormalizedCoordinates` is `VK_TRUE`, `anisotropyEnable` **must** be `VK_FALSE`
- If `unnormalizedCoordinates` is `VK_TRUE`, `compareEnable` **must** be `VK_FALSE`
- If any of `addressModeU`, `addressModeV` or `addressModeW` are `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`, `borderColor` **must** be a valid `VkBorderColor` value
- If `sampler Y'CBCR conversion` is enabled, `addressModeU`, `addressModeV`, and `addressModeW` **must** be `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`, `anisotropyEnable` **must** be `VK_FALSE`, and `unnormalizedCoordinates` **must** be `VK_FALSE`
- The sampler reduction mode **must** be set to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT` if sampler `Y'CBCR conversion` is enabled
- If the `VK_KHR_sampler_mirror_clamp_to_edge` extension is not enabled, `addressModeU`, `addressModeV` and `addressModeW` **must** not be `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE`
- If `compareEnable` is `VK_TRUE`, `compareOp` **must** be a valid `VkCompareOp` value
- If either `magFilter` or `minFilter` is `VK_FILTER_CUBIC_EXT`, `anisotropyEnable` **must** be `VK_FALSE`
- If `compareEnable` is `VK_TRUE`, the `reductionMode` member of `VkSamplerReductionModeCreateInfoEXT` **must** be `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT`
- If `flags` includes `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`, then `minFilter` and `magFilter` **must** be equal.
- If `flags` includes `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`, then `mipmapMode` **must** be `VK_SAMPLER_MIPMAP_MODE_NEAREST`.
- If `flags` includes `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`, then `minLod` and `maxLod` **must** be zero.

- If `flags` includes `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`, then `addressModeU` and `addressModeV` **must** each be either `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` or `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`.
- If `flags` includes `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`, then `anisotropyEnable` **must** be `VK_FALSE`.
- If `flags` includes `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`, then `compareEnable` **must** be `VK_FALSE`.
- If `flags` includes `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`, then `unnormalizedCoordinates` **must** be `VK_FALSE`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkSamplerReductionModeCreateInfoEXT` or `VkSamplerYcbcrConversionInfo`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be a valid combination of `VkSamplerCreateFlagBits` values
- `magFilter` **must** be a valid `VkFilter` value
- `minFilter` **must** be a valid `VkFilter` value
- `mipmapMode` **must** be a valid `VkSamplerMipmapMode` value
- `addressModeU` **must** be a valid `VkSamplerAddressMode` value
- `addressModeV` **must** be a valid `VkSamplerAddressMode` value
- `addressModeW` **must** be a valid `VkSamplerAddressMode` value

Bits which **can** be set in `VkSamplerCreateInfo::flags`, specifying additional parameters of a sampler, are:

```
typedef enum VkSamplerCreateFlagBits {
    VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT = 0x00000001,
    VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT = 0x00000002,
    VK_SAMPLER_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkSamplerCreateFlagBits;
```

- `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` specifies that the sampler will read from an image created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.
- `VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT` specifies that the implementation **may** use approximations when reconstructing a full color value for texture access from a subsampled image.

*Note*



The approximations used when `VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT` is specified are implementation defined. Some implementations **may** interpolate between fragment density levels in a subsampled image. In that case, this bit **may** be used to decide whether the interpolation factors are calculated per fragment or at a coarser granularity.

```
typedef VkFlags VkSamplerCreateFlags;
```

`VkSamplerCreateFlags` is a bitmask type for setting a mask of zero or more `VkSamplerCreateFlagBits`.

If the `pNext` chain of `VkSamplerCreateInfo` includes a `VkSamplerReductionModeCreateInfoEXT` structure, then that structure includes a mode that controls how texture filtering combines texel values.

The `VkSamplerReductionModeCreateInfoEXT` structure is defined as:

```
typedef struct VkSamplerReductionModeCreateInfoEXT {
    VkStructureType          sType;
    const void*              pNext;
    VkSamplerReductionModeEXT reductionMode;
} VkSamplerReductionModeCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `reductionMode` is a `VkSamplerReductionModeEXT` value controlling how texture filtering combines texel values.

If this structure is not present, `reductionMode` is considered to be `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO_EXT`
- `reductionMode` **must** be a valid `VkSamplerReductionModeEXT` value

Reduction modes are specified by `VkSamplerReductionModeEXT`, which takes values:

```
typedef enum VkSamplerReductionModeEXT {
    VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT = 0,
    VK_SAMPLER_REDUCTION_MODE_MIN_EXT = 1,
    VK_SAMPLER_REDUCTION_MODE_MAX_EXT = 2,
    VK_SAMPLER_REDUCTION_MODE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkSamplerReductionModeEXT;
```

- `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT` specifies that texel values are combined by computing a weighted average of values in the footprint, using weights as specified in [the image operations chapter](#).
- `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` specifies that texel values are combined by taking the component-wise minimum of values in the footprint with non-zero weights.
- `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` specifies that texel values are combined by taking the component-wise maximum of values in the footprint with non-zero weights.

Possible values of the `VkSamplerCreateInfo::magFilter` and `minFilter` parameters, specifying filters used for texture lookups, are:

```
typedef enum VkFilter {
    VK_FILTER_NEAREST = 0,
    VK_FILTER_LINEAR = 1,
    VK_FILTER_CUBIC_IMG = 1000015000,
    VK_FILTER_CUBIC_EXT = VK_FILTER_CUBIC_IMG,
    VK_FILTER_MAX_ENUM = 0x7FFFFFFF
} VkFilter;
```

- `VK_FILTER_NEAREST` specifies nearest filtering.
- `VK_FILTER_LINEAR` specifies linear filtering.
- `VK_FILTER_CUBIC_EXT` specifies cubic filtering.

These filters are described in detail in [Texel Filtering](#).

Possible values of the `VkSamplerCreateInfo::mipmapMode`, specifying the mipmap mode used for texture lookups, are:

```
typedef enum VkSamplerMipmapMode {
    VK_SAMPLER_MIPMAP_MODE_NEAREST = 0,
    VK_SAMPLER_MIPMAP_MODE_LINEAR = 1,
    VK_SAMPLER_MIPMAP_MODE_MAX_ENUM = 0x7FFFFFFF
} VkSamplerMipmapMode;
```

- `VK_SAMPLER_MIPMAP_MODE_NEAREST` specifies nearest filtering.
- `VK_SAMPLER_MIPMAP_MODE_LINEAR` specifies linear filtering.

These modes are described in detail in [Texel Filtering](#).

Possible values of the `VkSamplerCreateInfo::addressMode`\* parameters, specifying the behavior of sampling with coordinates outside the range [0,1] for the respective u, v, or w coordinate as defined in the [Wrapping Operation](#) section, are:

```
typedef enum VkSamplerAddressMode {
    VK_SAMPLER_ADDRESS_MODE_REPEAT = 0,
    VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT = 1,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE = 2,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER = 3,
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE = 4,
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE_KHR =
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE,
    VK_SAMPLER_ADDRESS_MODE_MAX_ENUM = 0x7FFFFFFF
} VkSamplerAddressMode;
```

- `VK_SAMPLER_ADDRESS_MODE_REPEAT` specifies that the repeat wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT` specifies that the mirrored repeat wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` specifies that the clamp to edge wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER` specifies that the clamp to border wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` specifies that the mirror clamp to edge wrap mode will be used. This is only valid if the `VK_KHR_sampler_mirror_clamp_to_edge` extension is enabled.

Possible values of `VkSamplerCreateInfo::borderColor`, specifying the border color used for texture lookups, are:

```
typedef enum VkBorderColor {
    VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK = 0,
    VK_BORDER_COLOR_INT_TRANSPARENT_BLACK = 1,
    VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK = 2,
    VK_BORDER_COLOR_INT_OPAQUE_BLACK = 3,
    VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE = 4,
    VK_BORDER_COLOR_INT_OPAQUE_WHITE = 5,
    VK_BORDER_COLOR_MAX_ENUM = 0x7FFFFFFF
} VkBorderColor;
```

- `VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK` specifies a transparent, floating-point format, black color.
- `VK_BORDER_COLOR_INT_TRANSPARENT_BLACK` specifies a transparent, integer format, black color.
- `VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK` specifies an opaque, floating-point format, black color.
- `VK_BORDER_COLOR_INT_OPAQUE_BLACK` specifies an opaque, integer format, black color.
- `VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE` specifies an opaque, floating-point format, white color.

- `VK_BORDER_COLOR_INT_OPAQUE_WHITE` specifies an opaque, integer format, white color.

These colors are described in detail in [Texel Replacement](#).

To destroy a sampler, call:

```
void vkDestroySampler(
    VkDevice                               device,
    VkSampler                             sampler,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the sampler.
- `sampler` is the sampler to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `sampler` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `sampler` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `sampler` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `sampler` is not `VK_NULL_HANDLE`, `sampler` **must** be a valid `VkSampler` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `sampler` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `sampler` **must** be externally synchronized

## 12.1. Sampler Y'CbCr conversion

To create a sampler with Y'CbCr conversion enabled, add a `VkSamplerYcbcrConversionInfo` to the `pNext` chain of the `VkSamplerCreateInfo` structure. To create a sampler Y'CbCr conversion, the `samplerYcbcrConversion` feature **must** be enabled. Conversion **must** be fixed at pipeline creation time, through use of a combined image sampler with an immutable sampler in

## `VkDescriptorSetLayoutBinding`.

A `VkSamplerYcbcrConversionInfo` **must** be provided for samplers to be used with image views that access `VK_IMAGE_ASPECT_COLOR_BIT` if the format appears in [Formats requiring sampler Y'CbCr conversion for VK\\_IMAGE\\_ASPECT\\_COLOR\\_BIT image views](#), or if the image view has an [external format](#).

The `VkSamplerYcbcrConversionInfo` structure is defined as:

```
typedef struct VkSamplerYcbcrConversionInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkSamplerYcbcrConversion conversion;
} VkSamplerYcbcrConversionInfo;
```

or the equivalent

```
typedef VkSamplerYcbcrConversionInfo VkSamplerYcbcrConversionInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `conversion` is a `VkSamplerYcbcrConversion` handle created with `vkCreateSamplerYcbcrConversion`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO`
- `conversion` **must** be a valid `VkSamplerYcbcrConversion` handle

A sampler Y'CbCr conversion is an opaque representation of a device-specific sampler Y'CbCr conversion description, represented as a `VkSamplerYcbcrConversion` handle:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSamplerYcbcrConversion)
```

or the equivalent

```
typedef VkSamplerYcbcrConversion VkSamplerYcbcrConversionKHR;
```

To create a `VkSamplerYcbcrConversion`, call:

```
VkResult vkCreateSamplerYcbcrConversion(
    VkDevice device,
    const VkSamplerYcbcrConversionCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSamplerYcbcrConversion* pYcbcrConversion);
```

or the equivalent command

```
VkResult vkCreateSamplerYcbcrConversionKHR(
    VkDevice device,
    const VkSamplerYcbcrConversionCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSamplerYcbcrConversion* pYcbcrConversion);
```

- **device** is the logical device that creates the sampler Y'CbCr conversion.
- **pCreateInfo** is a pointer to a `VkSamplerYcbcrConversionCreateInfo` structure specifying the requested sampler Y'CbCr conversion.
- **pAllocator** controls host memory allocation as described in the [Memory Allocation](#) chapter.
- **pYcbcrConversion** is a pointer to a `VkSamplerYcbcrConversion` handle in which the resulting sampler Y'CbCr conversion is returned.

The interpretation of the configured sampler Y'CbCr conversion is described in more detail in [the description of sampler Y'CbCr conversion](#) in the [Image Operations](#) chapter.

## Valid Usage

- The [sampler Y'CbCr conversion feature](#) **must** be enabled

## Valid Usage (Implicit)

- **device must** be a valid `VkDevice` handle
- **pCreateInfo must** be a valid pointer to a valid `VkSamplerYcbcrConversionCreateInfo` structure
- If **pAllocator** is not `NULL`, **pAllocator must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **pYcbcrConversion must** be a valid pointer to a `VkSamplerYcbcrConversion` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkSamplerYcbcrConversionCreateInfo` structure is defined as:

```
typedef struct VkSamplerYcbcrConversionCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkFormat                  format;
    VkSamplerYcbcrModelConversion  ycbcrModel;
    VkSamplerYcbcrRange        ycbcrRange;
    VkComponentMapping        components;
    VkChromaLocation           xChromaOffset;
    VkChromaLocation           yChromaOffset;
    VkFilter                   chromaFilter;
    VkBool32                  forceExplicitReconstruction;
} VkSamplerYcbcrConversionCreateInfo;
```

or the equivalent

```
typedef VkSamplerYcbcrConversionCreateInfo VkSamplerYcbcrConversionCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `format` is the format of the image from which color information will be retrieved.
- `ycbcrModel` describes the color matrix for conversion between color models.
- `ycbcrRange` describes whether the encoded values have headroom and foot room, or whether the encoding uses the full numerical range.
- `components` applies a *swizzle* based on `VkComponentSwizzle` enums prior to range expansion and color model conversion.
- `xChromaOffset` describes the *sample location* associated with downsampled chroma channels in the x dimension. `xChromaOffset` has no effect for formats in which chroma channels are the same resolution as the luma channel.
- `yChromaOffset` describes the *sample location* associated with downsampled chroma channels in the y dimension. `yChromaOffset` has no effect for formats in which the chroma channels are not downsampled vertically.
- `chromaFilter` is the filter for chroma reconstruction.

- `forceExplicitReconstruction` can be used to ensure that reconstruction is done explicitly, if supported.

*Note*



Setting `forceExplicitReconstruction` to `VK_TRUE` may have a performance penalty on implementations where explicit reconstruction is not the default mode of operation.

If the `pNext` chain has an instance of `VkExternalFormatANDROID` with non-zero `externalFormat` member, the sampler  $\text{Y}^{\text{C}_\text{B}} \text{C}_\text{R}$  conversion object represents an *external format conversion*, and `format` must be `VK_FORMAT_UNDEFINED`. Such conversions must only be used to sample image views with a matching `external format`. When creating an external format conversion, the value of `components` is ignored.

## Valid Usage

- If an external format conversion is being created, `format` must be `VK_FORMAT_UNDEFINED`, otherwise it must not be `VK_FORMAT_UNDEFINED`.
- `format` must support `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT` or `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`
- If the format does not support `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`, `xChromaOffset` and `yChromaOffset` must not be `VK_CHROMA_LOCATION_COSITED_EVEN`
- If the format does not support `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT`, `xChromaOffset` and `yChromaOffset` must not be `VK_CHROMA_LOCATION_MIDPOINT`
- `format` must represent unsigned normalized values (i.e. the format must be a `UNORM` format)
- If the format has a `_422` or `_420` suffix, then `components.g` must be `VK_COMPONENT_SWIZZLE_IDENTITY`
- If the format has a `_422` or `_420` suffix, then `components.a` must be `VK_COMPONENT_SWIZZLE_IDENTITY`, `VK_COMPONENT_SWIZZLE_ONE`, or `VK_COMPONENT_SWIZZLE_ZERO`
- If the format has a `_422` or `_420` suffix, then `components.r` must be `VK_COMPONENT_SWIZZLE_IDENTITY` or `VK_COMPONENT_SWIZZLE_B`
- If the format has a `_422` or `_420` suffix, then `components.b` must be `VK_COMPONENT_SWIZZLE_IDENTITY` or `VK_COMPONENT_SWIZZLE_R`
- If the format has a `_422` or `_420` suffix, and if either `components.r` or `components.b` is `VK_COMPONENT_SWIZZLE_IDENTITY`, both values must be `VK_COMPONENT_SWIZZLE_IDENTITY`
- If `ycbcrModel` is not `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`, then `components.r`, `components.g`, and `components.b` must correspond to channels of the `format`; that is, `components.r`, `components.g`, and `components.b` must not be `VK_COMPONENT_SWIZZLE_ZERO` or `VK_COMPONENT_SWIZZLE_ONE`, and must not correspond to a channel which contains zero or one as a consequence of conversion to `RGB`
- If `ycbcrRange` is `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW` then the R, G and B channels obtained by applying the component swizzle to `format` must each have a bit-depth greater than or equal to 8.
- If the format does not support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT`, `forceExplicitReconstruction` must be `FALSE`
- If the format does not support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT`, `chromaFilter` must be `VK_FILTER_NEAREST`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkExternalFormatANDROID`
- `format` **must** be a valid `VkFormat` value
- `ycbcrModel` **must** be a valid `VkSamplerYcbcrModelConversion` value
- `ycbcrRange` **must** be a valid `VkSamplerYcbcrRange` value
- `components` **must** be a valid `VkComponentMapping` structure
- `xChromaOffset` **must** be a valid `VkChromaLocation` value
- `yChromaOffset` **must** be a valid `VkChromaLocation` value
- `chromaFilter` **must** be a valid `VkFilter` value

If `chromaFilter` is `VK_FILTER_NEAREST`, chroma samples are reconstructed to luma channel resolution using nearest-neighbour sampling. Otherwise, chroma samples are reconstructed using interpolation. More details can be found in [the description of sampler Y'CbCr conversion](#) in the [Image Operations](#) chapter.

`VkSamplerYcbcrModelConversion` defines the conversion from the source color model to the shader color model. Possible values are:

```
typedef enum VkSamplerYcbcrModelConversion {
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY = 0,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY = 1,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709 = 2,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601 = 3,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020 = 4,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY_KHR =
VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY_KHR =
VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709_KHR =
VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601_KHR =
VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020_KHR =
VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020,
    VK_SAMPLER_YCBCR_MODEL_CONVERSION_MAX_ENUM = 0x7FFFFFFF
} VkSamplerYcbcrModelConversion;
```

or the equivalent

```
typedef VkSamplerYcbcrModelConversion VkSamplerYcbcrModelConversionKHR;
```

- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY` specifies that the input values to the conversion are unmodified.
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY` specifies no model conversion but the inputs are range expanded as for Y'CB<sub>R</sub>.
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709` specifies the color model conversion from Y'CB<sub>R</sub> to R'G'B' defined in BT.709 and described in the “BT.709 Y'CB<sub>R</sub> conversion” section of the [Kronos Data Format Specification](#).
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601` specifies the color model conversion from Y'CB<sub>R</sub> to R'G'B' defined in BT.601 and described in the “BT.601 Y'CB<sub>R</sub> conversion” section of the [Kronos Data Format Specification](#).
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020` specifies the color model conversion from Y'CB<sub>R</sub> to R'G'B' defined in BT.2020 and described in the “BT.2020 Y'CB<sub>R</sub> conversion” section of the [Kronos Data Format Specification](#).

In the `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_*` color models, for the input to the sampler Y'CB<sub>R</sub> range expansion and model conversion:

- the Y (Y' luma) channel corresponds to the G channel of an RGB image.
- the CB (C<sub>B</sub> or “U” blue color difference) channel corresponds to the B channel of an RGB image.
- the CR (C<sub>R</sub> or “V” red color difference) channel corresponds to the R channel of an RGB image.
- the alpha channel, if present, is not modified by color model conversion.

These rules reflect the mapping of channels after the channel swizzle operation (controlled by `VkSamplerYcbcrConversionCreateInfo::components`).

*Note*

For example, an “YUVA” 32-bit format comprising four 8-bit channels can be implemented as `VK_FORMAT_R8G8B8A8_UNORM` with a component mapping:



- `components.a = VK_COMPONENT_SWIZZLE_IDENTITY`
- `components.r = VK_COMPONENT_SWIZZLE_B`
- `components.g = VK_COMPONENT_SWIZZLE_R`
- `components.b = VK_COMPONENT_SWIZZLE_G`

The `VkSamplerYcbcrRange` enum describes whether color channels are encoded using the full range of numerical values or whether values are reserved for headroom and foot room. `VkSamplerYcbcrRange` is defined as:

```

typedef enum VkSamplerYcbcrRange {
    VK_SAMPLER_YCBCR_RANGE_ITU_FULL = 0,
    VK_SAMPLER_YCBCR_RANGE_ITU_NARROW = 1,
    VK_SAMPLER_YCBCR_RANGE_ITU_FULL_KHR = VK_SAMPLER_YCBCR_RANGE_ITU_FULL,
    VK_SAMPLER_YCBCR_RANGE_ITU_NARROW_KHR = VK_SAMPLER_YCBCR_RANGE_ITU_NARROW,
    VK_SAMPLER_YCBCR_RANGE_MAX_ENUM = 0x7FFFFFFF
} VkSamplerYcbcrRange;

```

or the equivalent

```
typedef VkSamplerYcbcrRange VkSamplerYcbcrRangeKHR;
```

- **VK\_SAMPLER\_YCBCR\_RANGE\_ITU\_FULL** specifies that the full range of the encoded values are valid and interpreted according to the ITU “full range” quantization rules.
- **VK\_SAMPLER\_YCBCR\_RANGE\_ITU\_NARROW** specifies that headroom and foot room are reserved in the numerical range of encoded values, and the remaining values are expanded according to the ITU “narrow range” quantization rules.

The formulae for these conversions is described in the [Sampler Y'CbCr Range Expansion](#) section of the [Image Operations](#) chapter.

No range modification takes place if `ycbcrModel` is **VK\_SAMPLER\_YCBCR\_MODEL\_CONVERSION\_RGB\_IDENTITY**; the `ycbcrRange` field of `VkSamplerYcbcrConversionCreateInfo` is ignored in this case.

The `VkChromaLocation` enum defines the location of downsampled chroma channel samples relative to the luma samples, and is defined as:

```

typedef enum VkChromaLocation {
    VK_CHROMA_LOCATION_COSITED_EVEN = 0,
    VK_CHROMA_LOCATION_MIDPOINT = 1,
    VK_CHROMA_LOCATION_COSITED_EVEN_KHR = VK_CHROMA_LOCATION_COSITED_EVEN,
    VK_CHROMA_LOCATION_MIDPOINT_KHR = VK_CHROMA_LOCATION_MIDPOINT,
    VK_CHROMA_LOCATION_MAX_ENUM = 0x7FFFFFFF
} VkChromaLocation;

```

or the equivalent

```
typedef VkChromaLocation VkChromaLocationKHR;
```

- **VK\_CHROMA\_LOCATION\_COSITED\_EVEN** specifies that downsampled chroma samples are aligned with luma samples with even coordinates.
- **VK\_CHROMA\_LOCATION\_MIDPOINT** specifies that downsampled chroma samples are located half way between each even luma sample and the nearest higher odd luma sample.

To destroy a sampler Y'CbCr conversion, call:

```
void vkDestroySamplerYcbcrConversion(  
    VkDevice device,  
    VkSamplerYcbcrConversion ycbcrConversion,  
    const VkAllocationCallbacks* pAllocator);
```

or the equivalent command

```
void vkDestroySamplerYcbcrConversionKHR(  
    VkDevice device,  
    VkSamplerYcbcrConversion ycbcrConversion,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the Y'CbCr conversion.
- `ycbcrConversion` is the conversion to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `ycbcrConversion` is not `VK_NULL_HANDLE`, `ycbcrConversion` **must** be a valid `VkSamplerYcbcrConversion` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `ycbcrConversion` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `ycbcrConversion` **must** be externally synchronized

# Chapter 13. Resource Descriptors

A *descriptor* is an opaque data structure representing a shader resource such as a buffer, buffer view, image view, sampler, or combined image sampler. Descriptors are organised into *descriptor sets*, which are bound during command recording for use in subsequent draw commands. The arrangement of content in each descriptor set is determined by a *descriptor set layout*, which determines what descriptors can be stored within it. The sequence of descriptor set layouts that **can** be used by a pipeline is specified in a *pipeline layout*. Each pipeline object **can** use up to `maxBoundDescriptorSets` (see [Limits](#)) descriptor sets.

Shaders access resources via variables decorated with a descriptor set and binding number that link them to a descriptor in a descriptor set. The shader interface mapping to bound descriptor sets is described in the [Shader Resource Interface](#) section.

Shaders **can** also access buffers without going through descriptors by using [Physical Storage Buffer Access](#) to access them through 64-bit addresses.

## 13.1. Descriptor Types

There are a number of different types of descriptor supported by Vulkan, corresponding to different resources or usage. The following sections describe the API definitions of each descriptor type. The mapping of each type to SPIR-V is listed in the [Shader Resource and Descriptor Type Correspondence](#) and [Shader Resource and Storage Class Correspondence](#) tables in the [Shader Interfaces](#) chapter.

### 13.1.1. Storage Image

A *storage image* (`VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`) is a descriptor type associated with an [image resource](#) via an [image view](#) that load, store, and atomic operations **can** be performed on.

Storage image loads are supported in all shader stages for image views whose [format features](#) contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`.

Stores to storage images are supported in compute shaders for image views whose [format features](#) contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`.

Atomic operations on storage images are supported in compute shaders for image views whose [format features](#) contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`.

When the `fragmentStoresAndAtomics` feature is enabled, stores and atomic operations are also supported for storage images in fragment shaders with the same set of image formats as supported in compute shaders. When the `vertexPipelineStoresAndAtomics` feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of image formats as supported in compute shaders.

The image subresources for a storage image **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` or `VK_IMAGE_LAYOUT_GENERAL` layout in order to access its data in a shader.

### 13.1.2. Sampler

A  *sampler descriptor*  (`VK_DESCRIPTOR_TYPE_SAMPLER`) is a descriptor type associated with a  *sampler*  object, used to control the behavior of  *sampling operations*  performed on a  *sampled image* .

### 13.1.3. Sampled Image

A  *sampled image*  (`VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`) is a descriptor type associated with an  *image resource*  via an  *image view*  that  *sampling operations*  **can** be performed on.

Shaders combine a  *sampled image*  variable and a  *sampler*  variable to perform sampling operations.

*Sampled images*  are supported in all shader stages for image views whose  *format features*  contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`.

The image subresources for a  *sampled image*  **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL` layout in order to access its data in a shader.

### 13.1.4. Combined Image Sampler

A  *combined image sampler*  (`VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`) is a single descriptor type associated with both a  *sampler*  and an  *image resource* , combining both a  *sampler*  and  *sampled image*  descriptor into a single descriptor.

If the descriptor refers to a  *sampler*  that performs  *Y'CbCr conversion*  or samples a  *subsampled image* , the  *sampler*  **must** only be used to sample the image in the same descriptor. Otherwise, the  *sampler*  and  *image*  in this type of descriptor **can** be used freely with any other samplers and images.

The image subresources for a  *combined image sampler*  **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL` layout in order to access its data in a shader.

*Note*



On some implementations, it **may** be more efficient to sample from an  *image*  using a combination of  *sampler*  and  *sampled image*  that are stored together in the descriptor set in a  *combined descriptor* .

### 13.1.5. Uniform Texel Buffer

A  *uniform texel buffer*  (`VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`) is a descriptor type associated with a  *buffer resource*  via a  *buffer view*  that  *formatted load operations*  **can** be performed on.

Uniform texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image.

Load operations from uniform texel buffers are supported in all shader stages for image formats which report support for the `VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

### 13.1.6. Storage Texel Buffer

A *storage texel buffer* (`VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`) is a descriptor type associated with a [buffer resource](#) via a [buffer view](#) that [formatted load, store, and atomic operations](#) **can** be performed on.

Storage texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image. Unlike [uniform texel buffers](#), these buffers can also be written to in the same way as for [storage images](#).

Storage texel buffer loads are supported in all shader stages for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

Stores to storage texel buffers are supported in compute shaders for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` feature via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

Atomic operations on storage texel buffers are supported in compute shaders for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT` feature via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

When the `fragmentStoresAndAtomics` feature is enabled, stores and atomic operations are also supported for storage texel buffers in fragment shaders with the same set of texel buffer formats as supported in compute shaders. When the `vertexPipelineStoresAndAtomics` feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of texel buffer formats as supported in compute shaders.

### 13.1.7. Storage Buffer

A *storage buffer* (`VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`) is a descriptor type associated with a [buffer resource](#) directly, described in a shader as a structure with various members that load, store, and atomic operations **can** be performed on.



#### Note

Atomic operations **can** only be performed on members of certain types as defined in the [SPIR-V environment appendix](#).

### 13.1.8. Uniform Buffer

A *uniform buffer* (`VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`) is a descriptor type associated with a [buffer resource](#) directly, described in a shader as a structure with various members that load operations

can be performed on.

### 13.1.9. Dynamic Uniform Buffer

A *dynamic uniform buffer* (`VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`) is almost identical to a [uniform buffer](#), and differs only in how the offset into the buffer is specified. The base offset calculated by the `VkDescriptorBufferInfo` when initially [updating the descriptor set](#) is added to a [dynamic offset](#) when binding the descriptor set.

### 13.1.10. Dynamic Storage Buffer

A *dynamic storage buffer* (`VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`) is almost identical to a [storage buffer](#), and differs only in how the offset into the buffer is specified. The base offset calculated by the `VkDescriptorBufferInfo` when initially [updating the descriptor set](#) is added to a [dynamic offset](#) when binding the descriptor set.

### 13.1.11. Inline Uniform Block

An *inline uniform block* (`VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`) is almost identical to a [uniform buffer](#), and differs only in taking its storage directly from the encompassing descriptor set instead of being backed by buffer memory. It is typically used to access a small set of constant data that does not require the additional flexibility provided by the indirection enabled when using a uniform buffer where the descriptor and the referenced buffer memory are decoupled. Compared to push constants, they allow reusing the same set of constant data across multiple disjoint sets of draw and dispatch commands.

Inline uniform block descriptors **cannot** be aggregated into arrays. Instead, the array size specified for an inline uniform block descriptor binding specifies the binding's capacity in bytes.

### 13.1.12. Input Attachment

An *input attachment* (`VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`) is a descriptor type associated with an [image resource](#) via an [image view](#) that **can** be used for [framebuffer local](#) load operations in fragment shaders.

All image formats that are supported for color attachments (`VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`) or depth/stencil attachments (`VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`) for a given image tiling mode are also supported for input attachments.

The image subresources for an input attachment **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_GENERAL` layout in order to access its data in a shader.

### 13.1.13. Acceleration Structure

An *acceleration structure* (`VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV`) is a descriptor type that is

used to retrieve scene geometry from within shaders bound to ray tracing pipelines. Shaders have read-only access to the memory.

## 13.2. Descriptor Sets

Descriptors are grouped together into descriptor set objects. A descriptor set object is an opaque object containing storage for a set of descriptors, where the types and number of descriptors is defined by a descriptor set layout. The layout object **may** be used to define the association of each descriptor binding with memory or other implementation resources. The layout is used both for determining the resources that need to be associated with the descriptor set, and determining the interface between shader stages and shader resources.

### 13.2.1. Descriptor Set Layout

A descriptor set layout object is defined by an array of zero or more descriptor bindings. Each individual descriptor binding is specified by a descriptor type, a count (array size) of the number of descriptors in the binding, a set of shader stages that **can** access the binding, and (if using immutable samplers) an array of sampler descriptors.

Descriptor set layout objects are represented by [VkDescriptorSetLayout](#) handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSetLayout)
```

To create descriptor set layout objects, call:

```
VkResult vkCreateDescriptorSetLayout(  
    VkDevice device,  
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkDescriptorSetLayout* pSetLayout);
```

- **device** is the logical device that creates the descriptor set layout.
- **pCreateInfo** is a pointer to a [VkDescriptorSetLayoutCreateInfo](#) structure specifying the state of the descriptor set layout object.
- **pAllocator** controls host memory allocation as described in the [Memory Allocation](#) chapter.
- **pSetLayout** is a pointer to a [VkDescriptorSetLayout](#) handle in which the resulting descriptor set layout object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkDescriptorSetLayoutCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `psetLayout` **must** be a valid pointer to a `VkDescriptorSetLayout` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Information about the descriptor set layout is passed in an instance of the `VkDescriptorSetLayoutCreateInfo` structure:

```
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDescriptorSetLayoutCreateFlags flags;
    uint32_t                        bindingCount;
    const VkDescriptorSetLayoutBinding* pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkDescriptorSetLayoutCreateFlagBits` specifying options for descriptor set layout creation.
- `bindingCount` is the number of elements in `pBindings`.
- `pBindings` is a pointer to an array of `VkDescriptorSetLayoutBinding` structures.

## Valid Usage

- The `VkDescriptorSetLayoutBinding::binding` members of the elements of the `pBindings` array **must** each have different values.
- If `flags` contains `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`, then all elements of `pBindings` **must** not have a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`
- If `flags` contains `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`, then all elements of `pBindings` **must** not have a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`
- If `flags` contains `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`, then the total number of elements of all bindings **must** be less than or equal to `VkPhysicalDevicePushDescriptorPropertiesKHR::maxPushDescriptors`
- If any binding has the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` bit set, `flags` **must** include `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT`
- If any binding has the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` bit set, then all bindings **must** not have `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkDescriptorSetLayoutBindingFlagsCreateInfoEXT`
- `flags` **must** be a valid combination of `VkDescriptorSetLayoutCreateFlagBits` values
- If `bindingCount` is not `0`, `pBindings` **must** be a valid pointer to an array of `bindingCount` valid `VkDescriptorSetLayoutBinding` structures

Bits which **can** be set in `VkDescriptorSetLayoutCreateInfo::flags` to specify options for descriptor set layout are:

```
typedef enum VkDescriptorSetLayoutCreateFlagBits {
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR = 0x00000001,
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT = 0x00000002,
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkDescriptorSetLayoutCreateFlagBits;
```

- `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR` specifies that descriptor sets **must** not be allocated using this layout, and descriptors are instead pushed by `vkCmdPushDescriptorSetKHR`.
- `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` specifies that descriptor sets using this layout **must** be allocated from a descriptor pool created with the

`VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` bit set. Descriptor set layouts created with this bit set have alternate limits for the maximum number of descriptors per-stage and per-pipeline layout. The `non-UpdateAfterBind` limits only count descriptors in sets created without this flag. The `UpdateAfterBind` limits count all descriptors, but the limits **may** be higher than the `non-UpdateAfterBind` limits.

```
typedef VkFlags VkDescriptorSetLayoutCreateFlags;
```

`VkDescriptorSetLayoutCreateFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorSetLayoutCreateFlagBits`.

The `VkDescriptorSetLayoutBinding` structure is defined as:

```
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t          binding;
    VkDescriptorType   descriptorType;
    uint32_t          descriptorCount;
    VkShaderStageFlags stageFlags;
    const VkSampler*   pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

- `binding` is the binding number of this entry and corresponds to a resource of the same binding number in the shader stages.
- `descriptorType` is a `VkDescriptorType` specifying which type of resource descriptors are used for this binding.
- `descriptorCount` is the number of descriptors contained in the binding, accessed in a shader as an array, except if `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` in which case `descriptorCount` is the size in bytes of the inline uniform block. If `descriptorCount` is zero this binding entry is reserved and the resource **must** not be accessed from any stage via this binding within any pipeline using the set layout.
- `stageFlags` member is a bitmask of `VkShaderStageFlagBits` specifying which pipeline shader stages **can** access a resource for this binding. `VK_SHADER_STAGE_ALL` is a shorthand specifying that all defined shader stages, including any additional stages defined by extensions, **can** access the resource.

If a shader stage is not included in `stageFlags`, then a resource **must** not be accessed from that stage via this binding within any pipeline using the set layout. Other than input attachments which are limited to the fragment shader, there are no limitations on what combinations of stages **can** use a descriptor binding, and in particular a binding **can** be used by both graphics stages and the compute stage.

- `pImmutableSamplers` affects initialization of samplers. If `descriptorType` specifies a `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` type descriptor, then `pImmutableSamplers` **can** be used to initialize a set of *immutable samplers*. Immutable samplers are permanently bound into the set layout and **must** not be changed; updating a `VK_DESCRIPTOR_TYPE_SAMPLER` descriptor with immutable samplers is not allowed and updates to a

`VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` descriptor with immutable samplers does not modify the samplers (the image views are updated, but the sampler updates are ignored). If `pImmutableSamplers` is not `NULL`, then it points to an array of sampler handles that will be copied into the set layout and used for the corresponding binding. Only the sampler handles are copied; the sampler objects **must** not be destroyed before the final use of the set layout and any descriptor pools and sets created using it. If `pImmutableSamplers` is `NULL`, then the sampler slots are dynamic and sampler handles **must** be bound into descriptor sets using this layout. If `descriptorType` is not one of these descriptor types, then `pImmutableSamplers` is ignored.

The above layout definition allows the descriptor bindings to be specified sparsely such that not all binding numbers between 0 and the maximum binding number need to be specified in the `pBindings` array. Bindings that are not specified have a `descriptorCount` and `stageFlags` of zero, and the value of `descriptorType` is undefined. However, all binding numbers between 0 and the maximum binding number in the `VkDescriptorSetLayoutCreateInfo::pBindings` array **may** consume memory in the descriptor set layout even if not all descriptor bindings are used, though it **should** not consume additional memory from the descriptor pool.

*Note*



The maximum binding number specified **should** be as compact as possible to avoid wasted memory.

## Valid Usage

- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `descriptorCount` is not `0` and `pImmutableSamplers` is not `NULL`, `pImmutableSamplers` **must** be a valid pointer to an array of `descriptorCount` valid `VkSampler` handles
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `descriptorCount` **must** be a multiple of `4`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `descriptorCount` **must** be less than or equal to `VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxInlineUniformBlockSize`
- If `descriptorCount` is not `0`, `stageFlags` **must** be a valid combination of `VkShaderStageFlagBits` values
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` and `descriptorCount` is not `0`, then `stageFlags` **must** be `0` or `VK_SHADER_STAGE_FRAGMENT_BIT`

## Valid Usage (Implicit)

- `descriptorType` **must** be a valid `VkDescriptorType` value

If the `pNext` chain of a `VkDescriptorSetLayoutCreateInfo` structure includes a `VkDescriptorSetLayoutBindingFlagsCreateInfoEXT` structure, then that structure includes an array of flags, one for each descriptor set layout binding.

The [VkDescriptorSetLayoutBindingFlagsCreateInfoEXT](#) structure is defined as:

```
typedef struct VkDescriptorSetLayoutBindingFlagsCreateInfoEXT {
    VkStructureType           sType;
    const void*                pNext;
    uint32_t                  bindingCount;
    const VkDescriptorBindingFlagsEXT*  pBindingFlags;
} VkDescriptorSetLayoutBindingFlagsCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `bindingCount` is zero or the number of elements in `pBindingFlags`.
- `pBindingFlags` is a pointer to an array of [VkDescriptorBindingFlagsEXT](#) bitfields, one for each descriptor set layout binding.

If `bindingCount` is zero or if this structure is not in the `pNext` chain, the [VkDescriptorBindingFlagsEXT](#) for each descriptor set layout binding is considered to be zero. Otherwise, the descriptor set layout binding at `VkDescriptorSetLayoutCreateInfo::pBindings[i]` uses the flags in `pBindingFlags[i]`.

## Valid Usage

- If `bindingCount` is not zero, `bindingCount` **must** equal `VkDescriptorSetLayoutCreateInfo`  
`::bindingCount`
- If `VkDescriptorSetLayoutCreateInfo::flags` includes  
`VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`, then all elements of  
`pBindingFlags` **must** not include `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`,  
`VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT`, or  
`VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT`
- If an element of `pBindingFlags` includes  
`VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT`, then all other elements of  
`VkDescriptorSetLayoutCreateInfo::pBindings` **must** have a smaller value of `binding`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`  
`::descriptorBindingUniformBufferUpdateAfterBind` is not enabled, all bindings with  
descriptor type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` **must** not use  
`VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`  
`::descriptorBindingSampledImageUpdateAfterBind` is not enabled, all bindings with  
descriptor type `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`,  
or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` **must** not use  
`VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`  
`::descriptorBindingStorageImageUpdateAfterBind` is not enabled, all bindings with  
descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` **must** not use  
`VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`  
`::descriptorBindingStorageBufferUpdateAfterBind` is not enabled, all bindings with  
descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` **must** not use  
`VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`  
`::descriptorBindingUniformTexelBufferUpdateAfterBind` is not enabled, all bindings with  
descriptor type `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` **must** not use  
`VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`  
`::descriptorBindingStorageTexelBufferUpdateAfterBind` is not enabled, all bindings with  
descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` **must** not use  
`VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- If `VkPhysicalDeviceInlineUniformBlockFeaturesEXT`  
`::descriptorBindingInlineUniformBlockUpdateAfterBind` is not enabled, all bindings with  
descriptor type `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` **must** not use  
`VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- All bindings with descriptor type `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`,  
`VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`  
**must** not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`

`::descriptorBindingUpdateUnusedWhilePending` is not enabled, all elements of `pBindingFlags` **must** not include `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT`

- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT::descriptorBindingPartiallyBound` is not enabled, all elements of `pBindingFlags` **must** not include `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT`
- If `VkPhysicalDeviceDescriptorIndexingFeaturesEXT::descriptorBindingVariableDescriptorCount` is not enabled, all elements of `pBindingFlags` **must** not include `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT`
- If an element of `pBindingFlags` includes `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT`, that element's `descriptorType` **must** not be `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO_EXT`
- If `bindingCount` is not `0`, and `pBindingFlags` is not `NULL`, `pBindingFlags` **must** be a valid pointer to an array of `bindingCount` valid combinations of `VkDescriptorBindingFlagBitsEXT` values

Bits which **can** be set in each element of `VkDescriptorSetLayoutBindingFlagsCreateInfoEXT` `::pBindingFlags` to specify options for the corresponding descriptor set layout binding are:

```
typedef enum VkDescriptorBindingFlagBitsEXT {
    VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT = 0x00000001,
    VK_DESCRIPTOR_BINDING_UPDATE_UNUSED WHILE_PENDING_BIT_EXT = 0x00000002,
    VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT = 0x00000004,
    VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT = 0x00000008,
    VK_DESCRIPTOR_BINDING_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDescriptorBindingFlagBitsEXT;
```

- `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` indicates that if descriptors in this binding are updated between when the descriptor set is bound in a command buffer and when that command buffer is submitted to a queue, then the submission will use the most recently set descriptors for this binding and the updates do not invalidate the command buffer. Descriptor bindings created with this flag are also partially exempt from the external synchronization requirement in `vkUpdateDescriptorSetWithTemplateKHR` and `vkUpdateDescriptorSets`. Multiple descriptors with this flag set **can** be updated concurrently in different threads, though the same descriptor **must** not be updated concurrently by two threads. Descriptors with this flag set **can** be updated concurrently with the set being bound to a command buffer in another thread, but not concurrently with the set being reset or freed.
- `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT` indicates that descriptors in this binding that are not *dynamically used* need not contain valid descriptors at the time the descriptors are consumed. A descriptor is dynamically used if any shader invocation executes an instruction

that performs any memory access using the descriptor.

- `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT` indicates that descriptors in this binding **can** be updated after a command buffer has bound this descriptor set, or while a command buffer that uses this descriptor set is pending execution, as long as the descriptors that are updated are not used by those command buffers. If `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT` is also set, then descriptors **can** be updated as long as they are not dynamically used by any shader invocations. If `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT` is not set, then descriptors **can** be updated as long as they are not statically used by any shader invocations.
- `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT` indicates that this descriptor binding has a variable size that will be specified when a descriptor set is allocated using this layout. The value of `descriptorCount` is treated as an upper bound on the size of the binding. This **must** only be used for the last binding in the descriptor set layout (i.e. the binding with the largest value of `binding`). For the purposes of counting against limits such as `maxDescriptorSet*` and `maxPerStageDescriptor*`, the full value of `descriptorCount` is counted, except for descriptor bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` where `descriptorCount` specifies the upper bound on the byte size of the binding, thus it counts against the `maxInlineUniformBlockSize` limit instead. .

*Note*

Note that while `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` and `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT` both involve updates to descriptor sets after they are bound, `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT` is a weaker requirement since it is only about descriptors that are not used, whereas `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` requires the implementation to observe updates to descriptors that are used.

```
typedef VkFlags VkDescriptorBindingFlagsEXT;
```

`VkDescriptorBindingFlagsEXT` is a bitmask type for setting a mask of zero or more `VkDescriptorBindingFlagBitsEXT`.

To query information about whether a descriptor set layout **can** be created, call:

```
void vkGetDescriptorSetLayoutSupport(  
    VkDevice device,  
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,  
    VkDescriptorSetLayoutSupport* pSupport);
```

or the equivalent command

```
void vkGetDescriptorSetLayoutSupportKHR(
    VkDevice device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,
    VkDescriptorSetLayoutSupport* pSupport);
```

- `device` is the logical device that would create the descriptor set layout.
- `pCreateInfo` is a pointer to a `VkDescriptorSetLayoutCreateInfo` structure specifying the state of the descriptor set layout object.
- `pSupport` is a pointer to a `VkDescriptorSetLayoutSupport` structure, in which information about support for the descriptor set layout object is returned.

Some implementations have limitations on what fits in a descriptor set which are not easily expressible in terms of existing limits like `maxDescriptorSet`\*, for example if all descriptor types share a limited space in memory but each descriptor is a different size or alignment. This command returns information about whether a descriptor set satisfies this limit. If the descriptor set layout satisfies the `VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors` limit, this command is guaranteed to return `VK_TRUE` in `VkDescriptorSetLayoutSupport::supported`. If the descriptor set layout exceeds the `VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors` limit, whether the descriptor set layout is supported is implementation-dependent and **may** depend on whether the descriptor sizes and alignments cause the layout to exceed an internal limit.

This command does not consider other limits such as `maxPerStageDescriptor`\*, and so a descriptor set layout that is supported according to this command **must** still satisfy the pipeline layout limits such as `maxPerStageDescriptor`\* in order to be used in a pipeline layout.

*Note*



This is a `VkDevice` query rather than `VkPhysicalDevice` because the answer **may** depend on enabled features.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkDescriptorSetLayoutCreateInfo` structure
- `pSupport` **must** be a valid pointer to a `VkDescriptorSetLayoutSupport` structure

Information about support for the descriptor set layout is returned in an instance of the `VkDescriptorSetLayoutSupport` structure:

```
typedef struct VkDescriptorSetLayoutSupport {
    VkStructureType sType;
    void* pNext;
    VkBool32 supported;
} VkDescriptorSetLayoutSupport;
```

or the equivalent

```
typedef VkDescriptorSetLayoutSupport VkDescriptorSetLayoutSupportKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `supported` specifies whether the descriptor set layout **can** be created.

`supported` is set to `VK_TRUE` if the descriptor set **can** be created, or else is set to `VK_FALSE`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkDescriptorSetVariableDescriptorCountLayoutSupportEXT`

If the `pNext` chain of a `VkDescriptorSetLayoutSupport` structure includes a `VkDescriptorSetVariableDescriptorCountLayoutSupportEXT` structure, then that structure returns additional information about whether the descriptor set layout is supported.

```
typedef struct VkDescriptorSetVariableDescriptorCountLayoutSupportEXT {  
    VkStructureType sType;  
    void* pNext;  
    uint32_t maxVariableDescriptorCount;  
} VkDescriptorSetVariableDescriptorCountLayoutSupportEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxVariableDescriptorCount` indicates the maximum number of descriptors supported in the highest numbered binding of the layout, if that binding is variable-sized. If the highest numbered binding of the layout has a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `maxVariableDescriptorCount` indicates the maximum byte size supported for the binding, if that binding is variable-sized.

If the create info includes a variable-sized descriptor, then `supported` is determined assuming the requested size of the variable-sized descriptor, and `maxVariableDescriptorCount` is set to the maximum size of that descriptor that **can** be successfully created (which is greater than or equal to the requested size passed in). If the create info does not include a variable-sized descriptor or if the `VkPhysicalDeviceDescriptorIndexingFeaturesEXT::descriptorBindingVariableDescriptorCount` feature is not enabled, then `maxVariableDescriptorCount` is set to zero. For the purposes of this command, a variable-sized descriptor binding with a `descriptorCount` of zero is treated as if the `descriptorCount` is one, and thus the binding is not ignored and the maximum descriptor count will be returned. If the layout is not supported, then the value written to `maxVariableDescriptorCount` is undefined.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT_EXT`

The following examples show a shader snippet using two descriptor sets, and application code that creates corresponding descriptor set layouts.

### *GLSL example*

```
//  
// binding to a single sampled image descriptor in set 0  
//  
layout (set=0, binding=0) uniform texture2D mySampledImage;  
  
//  
// binding to an array of sampled image descriptors in set 0  
//  
layout (set=0, binding=1) uniform texture2D myArrayOfSampledImages[12];  
  
//  
// binding to a single uniform buffer descriptor in set 1  
//  
layout (set=1, binding=0) uniform myUniformBuffer  
{  
    vec4 myElement[32];  
};
```

### SPIR-V example

```
...
%1 = OpExtInstImport "GLSL.std.450"
...
OpName %9 "mySampledImage"
OpName %14 "myArrayOfSampledImages"
OpName %18 "myUniformBuffer"
OpMemberName %18 0 "myElement"
OpName %20 ""
OpDecorate %9 DescriptorSet 0
OpDecorate %9 Binding 0
OpDecorate %14 DescriptorSet 0
OpDecorate %14 Binding 1
OpDecorate %17 ArrayStride 16
OpMemberDecorate %18 0 Offset 0
OpDecorate %18 Block
OpDecorate %20 DescriptorSet 1
OpDecorate %20 Binding 0
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 2D 0 0 0 1 Unknown
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
%10 = OpTypeInt 32 0
%11 = OpConstant %10 12
%12 = OpTypeArray %7 %11
%13 = OpTypePointer UniformConstant %12
%14 = OpVariable %13 UniformConstant
%15 = OpTypeVector %6 4
%16 = OpConstant %10 32
%17 = OpTypeArray %15 %16
%18 = OpTypeStruct %17
%19 = OpTypePointer Uniform %18
%20 = OpVariable %19 Uniform
...
```

### API example

```
VkResult myResult;

const VkDescriptorSetLayoutBinding myDescriptorSetLayoutBinding[] =
{
    // binding to a single image descriptor
    {
        0,                                     // binding
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,       // descriptorType
        1,                                     // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT,           // stageFlags
    }
};
```

```

        NULL                                // pImmutableSamplers
    },

    // binding to an array of image descriptors
    {
        1,                                     // binding
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,       // descriptorType
        12,                                    // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT,           // stageFlags
        NULL                                   // pImmutableSamplers
    },

    // binding to a single uniform buffer descriptor
    {
        0,                                     // binding
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,     // descriptorType
        1,                                     // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT,           // stageFlags
        NULL                                   // pImmutableSamplers
    }
};

const VkDescriptorSetLayoutCreateInfo myDescriptorSetLayoutCreateInfo[] =
{
    // Create info for first descriptor set with two descriptor bindings
    {
        VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,      // sType
        NULL,                                                 // pNext
        0,                                                   // flags
        2,                                                   // bindingCount
        &myDescriptorSetLayoutBinding[0]                         // pBindings
    },

    // Create info for second descriptor set with one descriptor binding
    {
        VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,      // sType
        NULL,                                                 // pNext
        0,                                                   // flags
        1,                                                   // bindingCount
        &myDescriptorSetLayoutBinding[2]                         // pBindings
    }
};

VkDescriptorSetLayout myDescriptorSetLayout[2];

// Create first descriptor set layout
// myResult = vkCreateDescriptorSetLayout(
//     myDevice,
//     &myDescriptorSetLayoutCreateInfo[0],

```

```

    NULL,
    &myDescriptorSetLayout[0]);

// Create second descriptor set layout
//
myResult = vkCreateDescriptorSetLayout(
    myDevice,
    &myDescriptorSetLayoutCreateInfo[1],
    NULL,
    &myDescriptorSetLayout[1]);

```

To destroy a descriptor set layout, call:

```

void vkDestroyDescriptorSetLayout(
    VkDevice                                     device,
    VkDescriptorSetLayout                         descriptorSetLayout,
    const VkAllocationCallbacks*                  pAllocator);

```

- `device` is the logical device that destroys the descriptor set layout.
- `descriptorSetLayout` is the descriptor set layout to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- If `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `descriptorSetLayout` is not `VK_NULL_HANDLE`, `descriptorSetLayout` **must** be a valid `VkDescriptorSetLayout` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `descriptorSetLayout` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `descriptorSetLayout` **must** be externally synchronized

### 13.2.2. Pipeline Layouts

Access to descriptor sets from a pipeline is accomplished through a *pipeline layout*. Zero or more descriptor set layouts and zero or more push constant ranges are combined to form a pipeline layout object describing the complete set of resources that **can** be accessed by a pipeline. The pipeline layout represents a sequence of descriptor sets with each having a specific layout. This sequence of layouts is used to determine the interface between shader stages and shader resources. Each pipeline is created using a pipeline layout.

Pipeline layout objects are represented by `VkPipelineLayout` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineLayout)
```

To create a pipeline layout, call:

```
VkResult vkCreatePipelineLayout(  
    VkDevice                                     device,  
    const VkPipelineLayoutCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks*     pAllocator,  
    VkPipelineLayout*               pPipelineLayout);
```

- `device` is the logical device that creates the pipeline layout.
- `pCreateInfo` is a pointer to a `VkPipelineLayoutCreateInfo` structure specifying the state of the pipeline layout object.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pPipelineLayout` is a pointer to a `VkPipelineLayout` handle in which the resulting pipeline layout object is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkPipelineLayoutCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pPipelineLayout` **must** be a valid pointer to a `VkPipelineLayout` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkPipelineLayoutCreateInfo` structure is defined as:

```
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineLayoutCreateFlags flags;
    uint32_t                  setLayoutCount;
    const VkDescriptorSetLayout* pSetLayouts;
    uint32_t                  pushConstantRangeCount;
    const VkPushConstantRange* pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `setLayoutCount` is the number of descriptor sets included in the pipeline layout.
- `pSetLayouts` is a pointer to an array of `VkDescriptorSetLayout` objects.
- `pushConstantRangeCount` is the number of push constant ranges included in the pipeline layout.
- `pPushConstantRanges` is a pointer to an array of `VkPushConstantRange` structures defining a set of push constant ranges for use in a single pipeline layout. In addition to descriptor set layouts, a pipeline layout also describes how many push constants **can** be accessed by each stage of the pipeline.

#### Note



Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

## Valid Usage

- `setLayoutCount` must be less than or equal to `VkPhysicalDeviceLimits::maxBoundDescriptorSets`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSamplers`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` and `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorUniformBuffers`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageBuffers`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSampledImages`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageImages`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorInputAttachments`
- The total number of bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxPerStageDescriptorInlineUniformBlocks`

- The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible to any given shader stage across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxPerStageDescriptorUpdateAfterBindSamplers`
- The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` and `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxPerStageDescriptorUpdateAfterBindUniformBuffers`
- The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxPerStageDescriptorUpdateAfterBindStorageBuffers`
- The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxPerStageDescriptorUpdateAfterBindSampledImages`
- The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxPerStageDescriptorUpdateAfterBindStorageImages`
- The total number of descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible to any given shader stage across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxPerStageDescriptorUpdateAfterBindInputAttachments`
- The total number of bindings with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` accessible to any given shader stage across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetSamplers`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to

### VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffers

- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffers`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffersDynamic`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetSampledImages`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetStorageImages`
- The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetInputAttachments`
- The total number of bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxDescriptorSetInlineUniformBlocks`
- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindSamples`
- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` accessible

across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to  
`VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindUniformBuffers`

- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindUniformBuffersDynamic`
- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindStorageBuffers`
- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindStorageBuffersDynamic`
- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindSampledImages`
- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindStorageImages`
- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceDescriptorIndexingPropertiesEXT::maxDescriptorSetUpdateAfterBindInputAttachments`
- The total number of bindings with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxDescriptorSetUpdateAfterBindInlineUniformBlocks`
- Any two elements of `pPushConstantRanges` **must** not include the same stage in `stageFlags`
- `pSetLayouts` **must** not contain more than one descriptor set layout that was created with `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR` set
- The total number of bindings with a `descriptorType` of `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV` accessible across all shader stages and across all elements of `pSetLayouts` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxDescriptorSetAccelerationStructures`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- If `setLayoutCount` is not `0`, `pSetLayouts` **must** be a valid pointer to an array of `setLayoutCount` valid `VkDescriptorSetLayout` handles
- If `pushConstantRangeCount` is not `0`, `pPushConstantRanges` **must** be a valid pointer to an array of `pushConstantRangeCount` valid `VkPushConstantRange` structures

```
typedef VkFlags VkPipelineLayoutCreateFlags;
```

`VkPipelineLayoutCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The `VkPushConstantRange` structure is defined as:

```
typedef struct VkPushConstantRange {  
    VkShaderStageFlags    stageFlags;  
    uint32_t              offset;  
    uint32_t              size;  
} VkPushConstantRange;
```

- `stageFlags` is a set of stage flags describing the shader stages that will access a range of push constants. If a particular stage is not included in the range, then accessing members of that range of push constants from the corresponding shader stage will return undefined values.
- `offset` and `size` are the start offset and size, respectively, consumed by the range. Both `offset` and `size` are in units of bytes and **must** be a multiple of 4. The layout of the push constant variables is specified in the shader.

## Valid Usage

- `offset` **must** be less than `VkPhysicalDeviceLimits::maxPushConstantsSize`
- `offset` **must** be a multiple of 4
- `size` **must** be greater than 0
- `size` **must** be a multiple of 4
- `size` **must** be less than or equal to `VkPhysicalDeviceLimits::maxPushConstantsSize` minus `offset`

## Valid Usage (Implicit)

- `stageFlags` **must** be a valid combination of `VkShaderStageFlagBits` values
- `stageFlags` **must** not be `0`

Once created, pipeline layouts are used as part of pipeline creation (see [Pipelines](#)), as part of binding descriptor sets (see [Descriptor Set Binding](#)), and as part of setting push constants (see [Push Constant Updates](#)). Pipeline creation accepts a pipeline layout as input, and the layout **may** be used to map (set, binding, arrayElement) tuples to implementation resources or memory locations within a descriptor set. The assignment of implementation resources depends only on the bindings defined in the descriptor sets that comprise the pipeline layout, and not on any shader source.

All resource variables [statically used](#) in all shaders in a pipeline **must** be declared with a (set,binding,arrayElement) that exists in the corresponding descriptor set layout and is of an appropriate descriptor type and includes the set of shader stages it is used by in `stageFlags`. The pipeline layout **can** include entries that are not used by a particular pipeline, or that are dead-code eliminated from any of the shaders. The pipeline layout allows the application to provide a consistent set of bindings across multiple pipeline compiles, which enables those pipelines to be compiled in a way that the implementation **may** cheaply switch pipelines without reprogramming the bindings.

Similarly, the push constant block declared in each shader (if present) **must** only place variables at offsets that are each included in a push constant range with `stageFlags` including the bit corresponding to the shader stage that uses it. The pipeline layout **can** include ranges or portions of ranges that are not used by a particular pipeline, or for which the variables have been dead-code eliminated from any of the shaders.

There is a limit on the total number of resources of each type that **can** be included in bindings in all descriptor set layouts in a pipeline layout as shown in [Pipeline Layout Resource Limits](#). The “Total Resources Available” column gives the limit on the number of each type of resource that **can** be included in bindings in all descriptor sets in the pipeline layout. Some resource types count against multiple limits. Additionally, there are limits on the total number of each type of resource that **can** be used in any pipeline stage as described in [Shader Resource Limits](#).

*Table 17. Pipeline Layout Resource Limits*

Total Resources Available	Resource Types
<code>maxDescriptorSetSamplers</code> or <code>maxDescriptorSetUpdateAfterBindSamplers</code>	sampler
	combined image sampler
<code>maxDescriptorSetSampledImages</code> or <code>maxDescriptorSetUpdateAfterBindSampledImages</code>	sampled image
	combined image sampler
	uniform texel buffer
<code>maxDescriptorSetStorageImages</code> or <code>maxDescriptorSetUpdateAfterBindStorageImages</code>	storage image
	storage texel buffer

Total Resources Available	Resource Types
<code>maxDescriptorSetUniformBuffers</code> or <code>maxDescriptorSetUpdateAfterBindUniformBuffers</code>	uniform buffer
<code>maxDescriptorSetUniformBuffersDynamic</code> or <code>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</code>	uniform buffer dynamic
<code>maxDescriptorSetStorageBuffers</code> or <code>maxDescriptorSetUpdateAfterBindStorageBuffers</code>	storage buffer
<code>maxDescriptorSetStorageBuffersDynamic</code> or <code>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</code>	storage buffer dynamic
<code>maxDescriptorSetInputAttachments</code> or <code>maxDescriptorSetUpdateAfterBindInputAttachments</code>	input attachment
<code>maxDescriptorSetInlineUniformBlocks</code> or <code>maxDescriptorSetUpdateAfterBindInlineUniformBlocks</code>	inline uniform block
<code>maxDescriptorSetAccelerationStructures</code>	acceleration structure

To destroy a pipeline layout, call:

```
void vkDestroyPipelineLayout(
    VkDevice device,
    VkPipelineLayout pipelineLayout,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the pipeline layout.
- `pipelineLayout` is the pipeline layout to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- If `VkAllocationCallbacks` were provided when `pipelineLayout` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `pipelineLayout` was created, `pAllocator` **must** be `NULL`
- `pipelineLayout` **must** not have been passed to any `vkCmd*` command for any command buffers that are still in the `recording` state when `vkDestroyPipelineLayout` is called

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `pipelineLayout` is not `VK_NULL_HANDLE`, `pipelineLayout` **must** be a valid `VkPipelineLayout` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `pipelineLayout` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `pipelineLayout` **must** be externally synchronized

## Pipeline Layout Compatibility

Two pipeline layouts are defined to be “compatible for `push constants`” if they were created with identical push constant ranges. Two pipeline layouts are defined to be “compatible for set N” if they were created with *identically defined* descriptor set layouts for sets zero through N, and if they were created with identical push constant ranges.

When binding a descriptor set (see [Descriptor Set Binding](#)) to set number N, if the previously bound descriptor sets for sets zero through N-1 were all bound using compatible pipeline layouts, then performing this binding does not disturb any of the lower numbered sets. If, additionally, the previous bound descriptor set for set N was bound using a pipeline layout compatible for set N, then the bindings in sets numbered greater than N are also not disturbed.

Similarly, when binding a pipeline, the pipeline **can** correctly access any previously bound descriptor sets which were bound with compatible pipeline layouts, as long as all lower numbered sets were also bound with compatible layouts.

Layout compatibility means that descriptor sets **can** be bound to a command buffer for use by any pipeline created with a compatible pipeline layout, and without having bound a particular pipeline first. It also means that descriptor sets **can** remain valid across a pipeline change, and the same resources will be accessible to the newly bound pipeline.

## Implementor’s Note

A consequence of layout compatibility is that when the implementation compiles a pipeline layout and maps pipeline resources to implementation resources, the mechanism for set N **should** only be a function of sets [0..N].

*Note*



Place the least frequently changing descriptor sets near the start of the pipeline layout, and place the descriptor sets representing the most frequently changing resources near the end. When pipelines are switched, only the descriptor set bindings that have been invalidated will need to be updated and the remainder of the descriptor set bindings will remain in place.

The maximum number of descriptor sets that **can** be bound to a pipeline layout is queried from physical device properties (see [maxBoundDescriptorSets](#) in [Limits](#)).

*API example*

```
const VkDescriptorSetLayout layouts[] = { layout1, layout2 };

const VkPushConstantRange ranges[] =
{
    {
        VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, // stageFlags
        0, // offset
        4 // size
    },
    {
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, // stageFlags
        4, // offset
        4 // size
    },
};

const VkPipelineLayoutCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO, // sType
    NULL, // pNext
    0, // flags
    2, // setLayoutCount
    layouts, // pSetLayouts
    2, // pushConstantRangeCount
    ranges // pPushConstantRanges
};

VkPipelineLayout myPipelineLayout;
myResult = vkCreatePipelineLayout(
    myDevice,
    &createInfo,
    NULL,
    &myPipelineLayout);
```

### 13.2.3. Allocation of Descriptor Sets

A *descriptor pool* maintains a pool of descriptors, from which descriptor sets are allocated. Descriptor pools are externally synchronized, meaning that the application **must** not allocate and/or free descriptor sets from the same pool in multiple threads simultaneously.

Descriptor pools are represented by `VkDescriptorPool` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorPool)
```

To create a descriptor pool object, call:

```
VkResult vkCreateDescriptorPool(  
    VkDevice device,  
    const VkDescriptorPoolCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkDescriptorPool* pDescriptorPool);
```

- `device` is the logical device that creates the descriptor pool.
- `pCreateInfo` is a pointer to a `VkDescriptorPoolCreateInfo` structure specifying the state of the descriptor pool object.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pDescriptorPool` is a pointer to a `VkDescriptorPool` handle in which the resulting descriptor pool object is returned.

`pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

The created descriptor pool is returned in `pDescriptorPool`.

#### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkDescriptorPoolCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pDescriptorPool` **must** be a valid pointer to a `VkDescriptorPool` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_FRAGMENTATION_EXT`

Additional information about the pool is passed in an instance of the `VkDescriptorPoolCreateInfo` structure:

```
typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType          sType;
    const void*               pNext;
    VkDescriptorPoolCreateFlags flags;
    uint32_t                  maxSets;
    uint32_t                  poolSizeCount;
    const VkDescriptorPoolSize* pPoolSizes;
} VkDescriptorPoolCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkDescriptorPoolCreateFlagBits` specifying certain supported operations on the pool.
- `maxSets` is the maximum number of descriptor sets that **can** be allocated from the pool.
- `poolSizeCount` is the number of elements in `pPoolSizes`.
- `pPoolSizes` is a pointer to an array of `VkDescriptorPoolSize` structures, each containing a descriptor type and number of descriptors of that type to be allocated in the pool.

If multiple `VkDescriptorPoolSize` structures appear in the `pPoolSizes` array then the pool will be created with enough storage for the total number of descriptors of each type.

Fragmentation of a descriptor pool is possible and **may** lead to descriptor set allocation failures. A failure due to fragmentation is defined as failing a descriptor set allocation despite the sum of all outstanding descriptor set allocations from the pool plus the requested allocation requiring no more than the total number of descriptors requested at pool creation. Implementations provide certain guarantees of when fragmentation **must** not cause allocation failure, as described below.

If a descriptor pool has not had any descriptor sets freed since it was created or most recently reset then fragmentation **must** not cause an allocation failure (note that this is always the case for a pool created without the `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` bit set). Additionally, if all sets allocated from the pool since it was created or most recently reset use the same number of descriptors (of each type) and the requested allocation also uses that same number of descriptors

(of each type), then fragmentation **must** not cause an allocation failure.

If an allocation failure occurs due to fragmentation, an application **can** create an additional descriptor pool to perform further descriptor set allocations.

If `flags` has the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` bit set, descriptor pool creation **may** fail with the error `VK_ERROR_FRAGMENTATION_EXT` if the total number of descriptors across all pools (including this one) created with this bit set exceeds `maxUpdateAfterBindDescriptorsInAllPools`, or if fragmentation of the underlying hardware resources occurs.

## Valid Usage

- `maxSets` **must** be greater than 0

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkDescriptorPoolInlineUniformBlockCreateInfoEXT`
- `flags` **must** be a valid combination of `VkDescriptorPoolCreateFlagBits` values
- `pPoolSizes` **must** be a valid pointer to an array of `poolSizeCount` valid `VkDescriptorPoolSize` structures
- `poolSizeCount` **must** be greater than 0

In order to be able to allocate descriptor sets having [inline uniform block](#) bindings the descriptor pool **must** be created with specifying the inline uniform block binding capacity of the descriptor pool, in addition to the total inline uniform data capacity in bytes which is specified through an instance of the `VkDescriptorPoolSize` structure with a `descriptorType` value of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`. This **can** be done by chaining an instance of the `VkDescriptorPoolInlineUniformBlockCreateInfoEXT` structure to the `pNext` chain of `VkDescriptorPoolCreateInfo`.

The `VkDescriptorPoolInlineUniformBlockCreateInfoEXT` structure is defined as:

```
typedef struct VkDescriptorPoolInlineUniformBlockCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t maxInlineUniformBlockBindings;
} VkDescriptorPoolInlineUniformBlockCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxInlineUniformBlockBindings` is the number of inline uniform block bindings to allocate.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO_EXT`

Bits which **can** be set in `VkDescriptorPoolCreateInfo::flags` to enable operations on a descriptor pool are:

```
typedef enum VkDescriptorPoolCreateFlagBits {
    VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT = 0x00000001,
    VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT = 0x00000002,
    VK_DESCRIPTOR_POOL_CREATE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkDescriptorPoolCreateFlagBits;
```

- `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` specifies that descriptor sets **can** return their individual allocations to the pool, i.e. all of `vkAllocateDescriptorSets`, `vkFreeDescriptorSets`, and `vkResetDescriptorPool` are allowed. Otherwise, descriptor sets allocated from the pool **must** not be individually freed back to the pool, i.e. only `vkAllocateDescriptorSets` and `vkResetDescriptorPool` are allowed.
- `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` specifies that descriptor sets allocated from this pool **can** include bindings with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` bit set. It is valid to allocate descriptor sets that have bindings that do not set the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` bit from a pool that has `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` set.

```
typedef VkFlags VkDescriptorPoolCreateFlags;
```

`VkDescriptorPoolCreateFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorPoolCreateFlagBits`.

The `VkDescriptorPoolSize` structure is defined as:

```
typedef struct VkDescriptorPoolSize {
    VkDescriptorType      type;
    uint32_t              descriptorCount;
} VkDescriptorPoolSize;
```

- `type` is the type of descriptor.
- `descriptorCount` is the number of descriptors of that type to allocate. If `type` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `descriptorCount` is the number of bytes to allocate for descriptors of this type.

*Note*



When creating a descriptor pool that will contain descriptors for combined image samplers of multi-planar formats, an application needs to account for non-trivial descriptor consumption when choosing the `descriptorCount` value, as indicated by `VkSamplerYcbcrConversionImageFormatProperties::combinedImageSamplerDescriptorCount`.

## Valid Usage

- `descriptorCount` **must** be greater than `0`
- If `type` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `descriptorCount` **must** be a multiple of `4`

## Valid Usage (Implicit)

- `type` **must** be a valid `VkDescriptorType` value

To destroy a descriptor pool, call:

```
void vkDestroyDescriptorPool(  
    VkDevice                                     device,  
    VkDescriptorPool                            descriptorPool,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the descriptor pool.
- `descriptorPool` is the descriptor pool to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

When a pool is destroyed, all descriptor sets allocated from the pool are implicitly freed and become invalid. Descriptor sets allocated from a given pool do not need to be freed before destroying that descriptor pool.

## Valid Usage

- All submitted commands that refer to `descriptorPool` (via any allocated descriptor sets) **must** have completed execution
- If `VkAllocationCallbacks` were provided when `descriptorPool` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `descriptorPool` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `descriptorPool` is not `VK_NULL_HANDLE`, `descriptorPool` **must** be a valid `VkDescriptorPool` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `descriptorPool` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `descriptorPool` **must** be externally synchronized

Descriptor sets are allocated from descriptor pool objects, and are represented by `VkDescriptorSet` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSet)
```

To allocate descriptor sets from a descriptor pool, call:

```
VkResult vkAllocateDescriptorSets(  
    VkDevice device,  
    const VkDescriptorSetAllocateInfo* pAllocateInfo,  
    VkDescriptorSet* pDescriptorSets);
```

- `device` is the logical device that owns the descriptor pool.
- `pAllocateInfo` is a pointer to a `VkDescriptorSetAllocateInfo` structure describing parameters of the allocation.
- `pDescriptorSets` is a pointer to an array of `VkDescriptorSet` handles in which the resulting descriptor set objects are returned.

The allocated descriptor sets are returned in `pDescriptorSets`.

When a descriptor set is allocated, the initial state is largely uninitialized and all descriptors are undefined. Descriptors also become undefined if the underlying resource is destroyed. Descriptor sets containing undefined descriptors **can** still be bound and used, subject to the following conditions:

- For descriptor set bindings created with the `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT` bit set, all descriptors in that binding that are dynamically used **must** have been populated before the descriptor set is **consumed**.
- For descriptor set bindings created without the `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT`

bit set, all descriptors in that binding that are statically used **must** have been populated before the descriptor set is **consumed**.

- Descriptor bindings with descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` **can** be undefined when the descriptor set is **consumed**; though values in that block will be undefined.
- Entries that are not used by a pipeline **can** have undefined descriptors.

If a call to `vkAllocateDescriptorSets` would cause the total number of descriptor sets allocated from the pool to exceed the value of `VkDescriptorPoolCreateInfo::maxSets` used to create `pAllocateInfo->descriptorPool`, then the allocation **may** fail due to lack of space in the descriptor pool. Similarly, the allocation **may** fail due to lack of space if the call to `vkAllocateDescriptorSets` would cause the number of any given descriptor type to exceed the sum of all the `descriptorCount` members of each element of `VkDescriptorPoolCreateInfo::pPoolSizes` with a `member` equal to that type.

Additionally, the allocation **may** also fail if a call to `vkAllocateDescriptorSets` would cause the total number of inline uniform block bindings allocated from the pool to exceed the value of `VkDescriptorPoolInlineUniformBlockCreateInfoEXT::maxInlineUniformBlockBindings` used to create the descriptor pool.

If the allocation fails due to no more space in the descriptor pool, and not because of system or device memory exhaustion, then `VK_ERROR_OUT_OF_POOL_MEMORY` **must** be returned.

`vkAllocateDescriptorSets` **can** be used to create multiple descriptor sets. If the creation of any of those descriptor sets fails, then the implementation **must** destroy all successfully created descriptor set objects from this command, set all entries of the `pDescriptorSets` array to `VK_NULL_HANDLE` and return the error.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pAllocateInfo` **must** be a valid pointer to a valid `VkDescriptorSetAllocateInfo` structure
- `pDescriptorSets` **must** be a valid pointer to an array of `pAllocateInfo::descriptorSetCount` `VkDescriptorSet` handles
- The value referenced by `pAllocateInfo::descriptorSetCount` **must** be greater than `0`

### Host Synchronization

- Host access to `pAllocateInfo::descriptorPool` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_FRAGMENTED_POOL`
- `VK_ERROR_OUT_OF_POOL_MEMORY`

The `VkDescriptorSetAllocateInfo` structure is defined as:

```
typedef struct VkDescriptorSetAllocateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkDescriptorPool           descriptorPool;
    uint32_t                  descriptorSetCount;
    const VkDescriptorSetLayout* pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `descriptorPool` is the pool which the sets will be allocated from.
- `descriptorSetCount` determines the number of descriptor sets to be allocated from the pool.
- `pSetLayouts` is a pointer to an array of descriptor set layouts, with each member specifying how the corresponding descriptor set is allocated.

## Valid Usage

- Each element of `pSetLayouts` must not have been created with `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR` set
- If any element of `pSetLayouts` was created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set, `descriptorPool` must have been created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` flag set

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkDescriptorSetVariableDescriptorCountAllocateInfoEXT`
- `descriptorPool` **must** be a valid `VkDescriptorPool` handle
- `pSetLayouts` **must** be a valid pointer to an array of `descriptorSetCount` valid `VkDescriptorSetLayout` handles
- `descriptorSetCount` **must** be greater than `0`
- Both of `descriptorPool`, and the elements of `pSetLayouts` **must** have been created, allocated, or retrieved from the same `VkDevice`

If the `pNext` chain of a `VkDescriptorSetAllocateInfo` structure includes a `VkDescriptorSetVariableDescriptorCountAllocateInfoEXT` structure, then that structure includes an array of descriptor counts for variable descriptor count bindings, one for each descriptor set being allocated.

The `VkDescriptorSetVariableDescriptorCountAllocateInfoEXT` structure is defined as:

```
typedef struct VkDescriptorSetVariableDescriptorCountAllocateInfoEXT {  
    VkStructureType sType;  
    const void* pNext;  
    uint32_t descriptorSetCount;  
    const uint32_t* pDescriptorCounts;  
} VkDescriptorSetVariableDescriptorCountAllocateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `descriptorSetCount` is zero or the number of elements in `pDescriptorCounts`.
- `pDescriptorCounts` is a pointer to an array of descriptor counts, with each member specifying the number of descriptors in a variable descriptor count binding in the corresponding descriptor set being allocated.

If `descriptorSetCount` is zero or this structure is not included in the `pNext` chain, then the variable lengths are considered to be zero. Otherwise, `pDescriptorCounts[i]` is the number of descriptors in the variable count descriptor binding in the corresponding descriptor set layout. If the variable count descriptor binding in the corresponding descriptor set layout has a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `pDescriptorCounts[i]` specifies the binding's capacity in bytes. If `VkDescriptorSetAllocateInfo::pSetLayouts[i]` does not include a variable count descriptor binding, then `pDescriptorCounts[i]` is ignored.

## Valid Usage

- If `descriptorSetCount` is not zero, `descriptorSetCount` **must** equal `VkDescriptorSetAllocateInfo::descriptorSetCount`
- If `VkDescriptorSetAllocateInfo::pSetLayouts[i]` has a variable descriptor count binding, then `pDescriptorCounts[i]` **must** be less than or equal to the descriptor count specified for that binding when the descriptor set layout was created.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO_EXT`
- If `descriptorSetCount` is not `0`, `pDescriptorCounts` **must** be a valid pointer to an array of `descriptorSetCount uint32_t` values

To free allocated descriptor sets, call:

```
VkResult vkFreeDescriptorSets(  
    VkDevice device,  
    VkDescriptorPool descriptorPool,  
    uint32_t descriptorSetCount,  
    const VkDescriptorSet* pDescriptorSets);
```

- `device` is the logical device that owns the descriptor pool.
- `descriptorPool` is the descriptor pool from which the descriptor sets were allocated.
- `descriptorSetCount` is the number of elements in the `pDescriptorSets` array.
- `pDescriptorSets` is a pointer to an array of handles to `VkDescriptorSet` objects.

After calling `vkFreeDescriptorSets`, all descriptor sets in `pDescriptorSets` are invalid.

## Valid Usage

- All submitted commands that refer to any element of `pDescriptorSets` **must** have completed execution
- `pDescriptorSets` **must** be a valid pointer to an array of `descriptorSetCount VkDescriptorSet` handles, each element of which **must** either be a valid handle or `VK_NULL_HANDLE`
- Each valid handle in `pDescriptorSets` **must** have been allocated from `descriptorPool`
- `descriptorPool` **must** have been created with the `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` flag

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `descriptorPool` **must** be a valid `VkDescriptorPool` handle
- `descriptorSetCount` **must** be greater than `0`
- `descriptorPool` **must** have been created, allocated, or retrieved from `device`
- Each element of `pDescriptorSets` that is a valid handle **must** have been created, allocated, or retrieved from `descriptorPool`

## Host Synchronization

- Host access to `descriptorPool` **must** be externally synchronized
- Host access to each member of `pDescriptorSets` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

To return all descriptor sets allocated from a given pool to the pool, rather than freeing individual descriptor sets, call:

```
VkResult vkResetDescriptorPool(  
    VkDevice                                     device,  
    VkDescriptorPool                            descriptorPool,  
    VkDescriptorPoolResetFlags                  flags);
```

- `device` is the logical device that owns the descriptor pool.
- `descriptorPool` is the descriptor pool to be reset.
- `flags` is reserved for future use.

Resetting a descriptor pool recycles all of the resources from all of the descriptor sets allocated from the descriptor pool back to the descriptor pool, and the descriptor sets are implicitly freed.

## Valid Usage

- All uses of `descriptorPool` (via any allocated descriptor sets) **must** have completed execution

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `descriptorPool` **must** be a valid `VkDescriptorPool` handle
- `flags` **must** be `0`
- `descriptorPool` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `descriptorPool` **must** be externally synchronized
- Host access to any `VkDescriptorSet` objects allocated from `descriptorPool` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

```
typedef VkFlags VkDescriptorPoolResetFlags;
```

`VkDescriptorPoolResetFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

### 13.2.4. Descriptor Set Updates

Once allocated, descriptor sets **can** be updated with a combination of write and copy operations. To update descriptor sets, call:

```
void vkUpdateDescriptorSets(  
    VkDevice                                     device,  
    uint32_t                                     descriptorWriteCount,  
    const VkWriteDescriptorSet*                  pDescriptorWrites,  
    uint32_t                                     descriptorCopyCount,  
    const VkCopyDescriptorSet*                  pDescriptorCopies);
```

- `device` is the logical device that updates the descriptor sets.
- `descriptorWriteCount` is the number of elements in the `pDescriptorWrites` array.
- `pDescriptorWrites` is a pointer to an array of `VkWriteDescriptorSet` structures describing the descriptor sets to write to.
- `descriptorCopyCount` is the number of elements in the `pDescriptorCopies` array.

- `pDescriptorCopies` is a pointer to an array of `VkCopyDescriptorSet` structures describing the descriptor sets to copy between.

The operations described by `pDescriptorWrites` are performed first, followed by the operations described by `pDescriptorCopies`. Within each array, the operations are performed in the order they appear in the array.

Each element in the `pDescriptorWrites` array describes an operation updating the descriptor set using descriptors for resources specified in the structure.

Each element in the `pDescriptorCopies` array is a `VkCopyDescriptorSet` structure describing an operation copying descriptors between sets.

If the `dstSet` member of any element of `pDescriptorWrites` or `pDescriptorCopies` is bound, accessed, or modified by any command that was recorded to a command buffer which is currently in the `recording` or `executable state`, and any of the descriptor bindings that are updated were not created with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` or `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT` bits set, that command buffer becomes invalid.

## Valid Usage

- Descriptor bindings updated by this command which were created without the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` or `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT` bits set **must** not be used by any command that was recorded to a command buffer which is in the `pending state`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `descriptorWriteCount` is not `0`, `pDescriptorWrites` **must** be a valid pointer to an array of `descriptorWriteCount` valid `VkWriteDescriptorSet` structures
- If `descriptorCopyCount` is not `0`, `pDescriptorCopies` **must** be a valid pointer to an array of `descriptorCopyCount` valid `VkCopyDescriptorSet` structures

## Host Synchronization

- Host access to `pDescriptorWrites[]`.`dstSet` **must** be externally synchronized
- Host access to `pDescriptorCopies[]`.`dstSet` **must** be externally synchronized

The `VkWriteDescriptorSet` structure is defined as:

```

typedef struct VkWriteDescriptorSet {
    VkStructureType           sType;
    const void*             pNext;
    VkDescriptorSet           dstSet;
    uint32_t                  dstBinding;
    uint32_t                  dstArrayElement;
    uint32_t                  descriptorCount;
    VkDescriptorType          descriptorType;
    const VkDescriptorImageInfo* pImageInfo;
    const VkDescriptorBufferInfo* pBufferInfo;
    const VkBufferView*       pTexelBufferView;
} VkWriteDescriptorSet;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **dstSet** is the destination descriptor set to update.
- **dstBinding** is the descriptor binding within that set.
- **dstArrayElement** is the starting element in that array. If the descriptor binding identified by **dstSet** and **dstBinding** has a descriptor type of **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT** then **dstArrayElement** specifies the starting byte offset within the binding.
- **descriptorCount** is the number of descriptors to update (the number of elements in **pImageInfo**, **pBufferInfo**, or **pTexelBufferView**, or a value matching the **dataSize** member of an instance of **VkWriteDescriptorSetInlineUniformBlockEXT** in the **pNext** chain, or a value matching the **accelerationStructureCount** of an instance of **VkWriteDescriptorSetAccelerationStructureNV** in the **pNext** chain). If the descriptor binding identified by **dstSet** and **dstBinding** has a descriptor type of **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT** then **descriptorCount** specifies the number of bytes to update.
- **descriptorType** is a **VkDescriptorType** specifying the type of each descriptor in **pImageInfo**, **pBufferInfo**, or **pTexelBufferView**, as described below. It **must** be the same type as that specified in **VkDescriptorSetLayoutBinding** for **dstSet** at **dstBinding**. The type of the descriptor also controls which array the descriptors are taken from.
- **pImageInfo** is a pointer to an array of **VkDescriptorImageInfo** structures or is ignored, as described below.
- **pBufferInfo** is a pointer to an array of **VkDescriptorBufferInfo** structures or is ignored, as described below.
- **pTexelBufferView** is a pointer to an array of **VkBufferView** handles as described in the **Buffer Views** section or is ignored, as described below.

Only one of **pImageInfo**, **pBufferInfo**, or **pTexelBufferView** members is used according to the descriptor type specified in the **descriptorType** member of the containing **VkWriteDescriptorSet** structure, or none of them in case **descriptorType** is **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT**, in which case the source data for the descriptor writes is taken from the instance of **VkWriteDescriptorSetInlineUniformBlockEXT** in the **pNext** chain of **VkWriteDescriptorSet**, or if **descriptorType** is **VK\_DESCRIPTOR\_TYPE\_ACCELERATION\_STRUCTURE\_NV**, in which case the source data for

the descriptor writes is taken from the instance of `VkWriteDescriptorSetAccelerationStructureNV` in the `pNext` chain of `VkWriteDescriptorSet`, as specified below.

If the `dstBinding` has fewer than `descriptorCount` array elements remaining starting from `dstArrayElement`, then the remainder will be used to update the subsequent binding - `dstBinding+1` starting at array element zero. If a binding has a `descriptorCount` of zero, it is skipped. This behavior applies recursively, with the update affecting consecutive bindings as needed to update all `descriptorCount` descriptors.

*Note*

The same behavior applies to bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` where `descriptorCount` specifies the number of bytes to update while `dstArrayElement` specifies the starting byte offset, thus in this case if the `dstBinding` has a smaller byte size than the sum of `dstArrayElement` and `descriptorCount`, then the remainder will be used to update the subsequent binding - `dstBinding+1` starting at offset zero. This falls out as a special case of the above rule.



## Valid Usage

- `dstBinding` **must** be less than or equal to the maximum value of `binding` of all `VkDescriptorSetLayoutBinding` structures specified when `dstSet`'s descriptor set layout was created
- `dstBinding` **must** be a binding with a non-zero `descriptorCount`
- All consecutive bindings updated via a single `VkWriteDescriptorSet` structure, except those with a `descriptorCount` of zero, **must** have identical `descriptorType` and `stageFlags`.
- All consecutive bindings updated via a single `VkWriteDescriptorSet` structure, except those with a `descriptorCount` of zero, **must** all either use immutable samplers or **must** all not use immutable samplers.
- `descriptorType` **must** match the type of `dstBinding` within `dstSet`
- `dstSet` **must** be a valid `VkDescriptorSet` handle
- The sum of `dstArrayElement` and `descriptorCount` **must** be less than or equal to the number of array elements in the descriptor set binding specified by `dstBinding`, and all applicable consecutive bindings, as described by `consecutive binding updates`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, `dstArrayElement` **must** be an integer multiple of 4
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, `descriptorCount` **must** be an integer multiple of 4
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, `pImageInfo` **must** be a valid pointer to an array of `descriptorCount` valid `VkDescriptorImageInfo` structures
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, `pTexelBufferView` **must** be a valid pointer to an array of `descriptorCount` valid `VkBufferView` handles
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, `pBufferInfo` **must** be a valid pointer to an array of `descriptorCount` valid `VkDescriptorBufferInfo` structures
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `dstSet` was not allocated with a layout that included immutable samplers for `dstBinding` with `descriptorType`, the `sampler` member of each element of `pImageInfo` **must** be a valid `VkSampler` object
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` and `imageLayout` members of each element of `pImageInfo` **must** be a valid `VkImageView` and `VkImageLayout`, respectively
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, the `pNext` chain **must** include a `VkWriteDescriptorSetInlineUniformBlockEXT` structure whose `dataSize`

member equals `descriptorCount`

- If `descriptorType` is `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV`, the `pNext` chain **must** include a `VkWriteDescriptorSetAccelerationStructureNV` structure whose `accelerationStructureCount` member equals `descriptorCount`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, then the `imageView` member of each `pImageInfo` element **must** have been created without a `VkSamplerYcbcrConversionInfo` structure in its `pNext` chain
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and if any element of `pImageInfo` has a `imageView` member that was created with a `VkSamplerYcbcrConversionInfo` structure in its `pNext` chain, then `dstSet` **must** have been allocated with a layout that included immutable samplers for `dstBinding`, and the corresponding immutable sampler **must** have been created with an *identically defined* `VkSamplerYcbcrConversionInfo` object
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `dstSet` was allocated with a layout that included immutable samplers for `dstBinding`, then the `imageView` member of each element of `pImageInfo` which corresponds to an immutable sampler that enables sampler Y'CbCr conversion **must** have been created with a `VkSamplerYcbcrConversionInfo` structure in its `pNext` chain with an *identically defined* `VkSamplerYcbcrConversionInfo` to the corresponding immutable sampler
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, for each descriptor that will be accessed via load or store operations the `imageLayout` member for corresponding elements of `pImageInfo` **must** be `VK_IMAGE_LAYOUT_GENERAL`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `offset` member of each element of `pBufferInfo` **must** be a multiple of `VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `offset` member of each element of `pBufferInfo` **must** be a multiple of `VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, and the `buffer` member of any element of `pBufferInfo` is the handle of a non-sparse buffer, then that buffer **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `buffer` member of each element of `pBufferInfo` **must** have been created with `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` set
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `buffer` member of each element of `pBufferInfo` **must** have been created with `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT` set
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, the `range` member of each element of `pBufferInfo`, or the effective range if `range` is `VK_WHOLE_SIZE`, **must** be less than or equal to

### VkPhysicalDeviceLimits::maxUniformBufferRange

- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, the `range` member of each element of `pBufferInfo`, or the effective range if `range` is `VK_WHOLE_SIZE`, **must** be less than or equal to `VkPhysicalDeviceLimits::maxStorageBufferRange`
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`, the `VkBuffer` that each element of `pTexelBufferView` was created from **must** have been created with `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` set
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, the `VkBuffer` that each element of `pTexelBufferView` was created from **must** have been created with `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` set
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` **must** have been created with the identity swizzle
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `imageView` member of each element of `pImageInfo` **must** have been created with `VK_IMAGE_USAGE_SAMPLED_BIT` set
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `imageLayout` member of each element of `pImageInfo` **must** be a member of the list given in [Sampled Image](#) or [Combined Image Sampler](#), corresponding to its type
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` **must** have been created with `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` set
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, the `imageView` member of each element of `pImageInfo` **must** have been created with `VK_IMAGE_USAGE_STORAGE_BIT` set
- All consecutive bindings updated via a single `VkWriteDescriptorSet` structure, except those with a `descriptorCount` of zero, **must** have identical `VkDescriptorBindingFlagBitsEXT`.
- If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER`, then `dstSet` **must** not have been allocated with a layout that included immutable samplers for `dstBinding`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkWriteDescriptorSetAccelerationStructureNV` or `VkWriteDescriptorSetInlineUniformBlockEXT`
- Each `sType` member in the `pNext` chain must be unique
- `descriptorType` must be a valid `VkDescriptorType` value
- `descriptorCount` must be greater than `0`
- Both of `dstSet`, and the elements of `pTexelBufferView` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

The type of descriptors in a descriptor set is specified by `VkWriteDescriptorSet::descriptorType`, which must be one of the values:

```
typedef enum VkDescriptorType {
    VK_DESCRIPTOR_TYPE_SAMPLER = 0,
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER = 1,
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE = 2,
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE = 3,
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER = 4,
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER = 5,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER = 6,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER = 7,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC = 8,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC = 9,
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT = 10,
    VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT = 1000138000,
    VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV = 1000165000,
    VK_DESCRIPTOR_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkDescriptorType;
```

- `VK_DESCRIPTOR_TYPE_SAMPLER` specifies a [sampler descriptor](#).
- `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` specifies a [combined image sampler descriptor](#).
- `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` specifies a [sampled image descriptor](#).
- `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` specifies a [storage image descriptor](#).
- `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` specifies a [uniform texel buffer descriptor](#).
- `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` specifies a [storage texel buffer descriptor](#).
- `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` specifies a [uniform buffer descriptor](#).
- `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` specifies a [storage buffer descriptor](#).
- `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` specifies a [dynamic uniform buffer descriptor](#).

- `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` specifies a [dynamic storage buffer descriptor](#).
- `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` specifies an [input attachment descriptor](#).
- `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` specifies an [inline uniform block](#).

When a descriptor set is updated via elements of `VkWriteDescriptorSet`, members of `pImageInfo`, `pBufferInfo` and `pTexelBufferView` are only accessed by the implementation when they correspond to descriptor type being defined - otherwise they are ignored. The members accessed are as follows for each descriptor type:

- For `VK_DESCRIPTOR_TYPE_SAMPLER`, only the `sampler` member of each element of `VkWriteDescriptorSet::pImageInfo` is accessed.
- For `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, only the `imageView` and `imageLayout` members of each element of `VkWriteDescriptorSet::pImageInfo` are accessed.
- For `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, all members of each element of `VkWriteDescriptorSet::pImageInfo` are accessed.
- For `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, all members of each element of `VkWriteDescriptorSet::pBufferInfo` are accessed.
- For `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, each element of `VkWriteDescriptorSet::pTexelBufferView` is accessed.

When updating descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, none of the `pImageInfo`, `pBufferInfo`, or `pTexelBufferView` members are accessed, instead the source data of the descriptor update operation is taken from the instance of `VkWriteDescriptorSetInlineUniformBlockEXT` in the `pNext` chain of `VkWriteDescriptorSet`. When updating descriptors with a `descriptorType` of `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV`, none of the `pImageInfo`, `pBufferInfo`, or `pTexelBufferView` members are accessed, instead the source data of the descriptor update operation is taken from the instance of `VkWriteDescriptorSetAccelerationStructureNV` in the `pNext` chain of `VkWriteDescriptorSet`.

The `VkDescriptorBufferInfo` structure is defined as:

```
typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset;
    VkDeviceSize    range;
} VkDescriptorBufferInfo;
```

- `buffer` is the buffer resource.
- `offset` is the offset in bytes from the start of `buffer`. Access to buffer memory via this descriptor uses addressing that is relative to this starting offset.
- `range` is the size in bytes that is used for this descriptor update, or `VK_WHOLE_SIZE` to use the range from `offset` to the end of the buffer.

*Note*



When setting `range` to `VK_WHOLE_SIZE`, the effective range **must** not be larger than the maximum range for the descriptor type (`maxUniformBufferRange` or `maxStorageBufferRange`). This means that `VK_WHOLE_SIZE` is not typically useful in the common case where uniform buffer descriptors are suballocated from a buffer that is much larger than `maxUniformBufferRange`.

For `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` descriptor types, `offset` is the base offset from which the dynamic offset is applied and `range` is the static size used for all dynamic offsets.

### Valid Usage

- `offset` **must** be less than the size of `buffer`
- If `range` is not equal to `VK_WHOLE_SIZE`, `range` **must** be greater than `0`
- If `range` is not equal to `VK_WHOLE_SIZE`, `range` **must** be less than or equal to the size of `buffer` minus `offset`

### Valid Usage (Implicit)

- `buffer` **must** be a valid `VkBuffer` handle

The `VkDescriptorImageInfo` structure is defined as:

```
typedef struct VkDescriptorImageInfo {  
    VkSampler      sampler;  
    VkImageView    imageView;  
    VkImageLayout  imageLayout;  
} VkDescriptorImageInfo;
```

- `sampler` is a sampler handle, and is used in descriptor updates for types `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` if the binding being updated does not use immutable samplers.
- `imageView` is an image view handle, and is used in descriptor updates for types `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`.
- `imageLayout` is the layout that the image subresources accessible from `imageView` will be in at the time this descriptor is accessed. `imageLayout` is used in descriptor updates for types `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`.

Members of `VkDescriptorImageInfo` that are not used in an update (as described above) are ignored.

## Valid Usage

- `imageView` **must** not be 2D or 2D array image view created from a 3D image
- If `imageView` is created from a depth/stencil image, the `aspectMask` used to create the `imageView` **must** include either `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` but not both.
- `imageLayout` **must** match the actual `VkImageLayout` of each subresource accessible from `imageView` at the time this descriptor is accessed as defined by the `image layout matching rules`
- If `sampler` is used and the `VkFormat` of the image is a `multi-planar format`, the image **must** have been created with `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, and the `aspectMask` of the `imageView` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or (for three-plane formats only) `VK_IMAGE_ASPECT_PLANE_2_BIT`

## Valid Usage (Implicit)

- Both of `imageView`, and `sampler` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

If the `descriptorType` member of `VkWriteDescriptorSet` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then the data to write to the descriptor set is specified through an instance of `VkWriteDescriptorSetInlineUniformBlockEXT` chained to the `pNext` chain of `VkWriteDescriptorSet`.

The `VkWriteDescriptorSetInlineUniformBlockEXT` structure is defined as:

```
typedef struct VkWriteDescriptorSetInlineUniformBlockEXT {
    VkStructureType sType;
    const void*     pNext;
    uint32_t        dataSize;
    const void*     pData;
} VkWriteDescriptorSetInlineUniformBlockEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `dataSize` is the number of bytes of inline uniform block data pointed to by `pData`.
- `pData` is a pointer to `dataSize` number of bytes of data to write to the inline uniform block.

## Valid Usage

- `dataSize` **must** be an integer multiple of 4

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT`
- `pData` **must** be a valid pointer to an array of `dataSize` bytes
- `dataSize` **must** be greater than `0`

The `VkWriteDescriptorSetAccelerationStructureNV` structure is defined as:

```
typedef struct VkWriteDescriptorSetAccelerationStructureNV {  
    VkStructureType           sType;  
    const void*                pNext;  
    uint32_t                  accelerationStructureCount;  
    const VkAccelerationStructureNV* pAccelerationStructures;  
} VkWriteDescriptorSetAccelerationStructureNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `accelerationStructureCount` is the number of elements in `pAccelerationStructures`.
- `pAccelerationStructures` are the acceleration structures to update.

## Valid Usage

- `accelerationStructureCount` **must** be equal to `descriptorCount` in the extended structure
- Each acceleration structure in `pAccelerationStructures` must have been created with `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_NV`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_NV`
- `pAccelerationStructures` **must** be a valid pointer to an array of `accelerationStructureCount` valid `VkAccelerationStructureNV` handles
- `accelerationStructureCount` **must** be greater than `0`

The `VkCopyDescriptorSet` structure is defined as:

```

typedef struct VkCopyDescriptorSet {
    VkStructureType      sType;
    const void*        pNext;
    VkDescriptorSet      srcSet;
    uint32_t             srcBinding;
    uint32_t             srcArrayElement;
    VkDescriptorSet      dstSet;
    uint32_t             dstBinding;
    uint32_t             dstArrayElement;
    uint32_t             descriptorCount;
} VkCopyDescriptorSet;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **srcSet**, **srcBinding**, and **srcArrayElement** are the source set, binding, and array element, respectively. If the descriptor binding identified by **srcSet** and **srcBinding** has a descriptor type of **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT** then **srcArrayElement** specifies the starting byte offset within the binding to copy from.
- **dstSet**, **dstBinding**, and **dstArrayElement** are the destination set, binding, and array element, respectively. If the descriptor binding identified by **dstSet** and **dstBinding** has a descriptor type of **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT** then **dstArrayElement** specifies the starting byte offset within the binding to copy to.
- **descriptorCount** is the number of descriptors to copy from the source to destination. If **descriptorCount** is greater than the number of remaining array elements in the source or destination binding, those affect consecutive bindings in a manner similar to **VkWriteDescriptorSet** above. If the descriptor binding identified by **srcSet** and **srcBinding** has a descriptor type of **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT** then **descriptorCount** specifies the number of bytes to copy and the remaining array elements in the source or destination binding refer to the remaining number of bytes in those.

## Valid Usage

- `srcBinding` **must** be a valid binding within `srcSet`
- The sum of `srcArrayElement` and `descriptorCount` **must** be less than or equal to the number of array elements in the descriptor set binding specified by `srcBinding`, and all applicable consecutive bindings, as described by [consecutive binding updates](#)
- `dstBinding` **must** be a valid binding within `dstSet`
- The sum of `dstArrayElement` and `descriptorCount` **must** be less than or equal to the number of array elements in the descriptor set binding specified by `dstBinding`, and all applicable consecutive bindings, as described by [consecutive binding updates](#)
- The type of `dstBinding` within `dstSet` **must** be equal to the type of `srcBinding` within `srcSet`
- If `srcSet` is equal to `dstSet`, then the source and destination ranges of descriptors **must** not overlap, where the ranges **may** include array elements from consecutive bindings as described by [consecutive binding updates](#)
- If the descriptor type of the descriptor set binding specified by `srcBinding` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, `srcArrayElement` **must** be an integer multiple of 4
- If the descriptor type of the descriptor set binding specified by `dstBinding` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, `dstArrayElement` **must** be an integer multiple of 4
- If the descriptor type of the descriptor set binding specified by either `srcBinding` or `dstBinding` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, `descriptorCount` **must** be an integer multiple of 4
- If `srcSet`'s layout was created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` flag set, then `dstSet`'s layout **must** also have been created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` flag set
- If `srcSet`'s layout was created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` flag set, then `dstSet`'s layout **must** also have been created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` flag set
- If the descriptor pool from which `srcSet` was allocated was created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` flag set, then the descriptor pool from which `dstSet` was allocated **must** also have been created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` flag set
- If the descriptor pool from which `srcSet` was allocated was created without the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` flag set, then the descriptor pool from which `dstSet` was allocated **must** also have been created without the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` flag set
- If the descriptor type of the descriptor set binding specified by `dstBinding` is `VK_DESCRIPTOR_TYPE_SAMPLER`, then `dstSet` **must** not have been allocated with a layout that included immutable samplers for `dstBinding`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET`
- `pNext` **must** be `NULL`
- `srcSet` **must** be a valid `VkDescriptorSet` handle
- `dstSet` **must** be a valid `VkDescriptorSet` handle
- Both of `dstSet`, and `srcSet` **must** have been created, allocated, or retrieved from the same `VkDevice`

### 13.2.5. Descriptor Update Templates

A descriptor update template specifies a mapping from descriptor update information in host memory to descriptors in a descriptor set. It is designed to avoid passing redundant information to the driver when frequently updating the same set of descriptors in descriptor sets.

Descriptor update template objects are represented by `VkDescriptorUpdateTemplate` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorUpdateTemplate)
```

or the equivalent

```
typedef VkDescriptorUpdateTemplate VkDescriptorUpdateTemplateKHR;
```

### 13.2.6. Descriptor Set Updates with Templates

Updating a large `VkDescriptorSet` array **can** be an expensive operation since an application **must** specify one `VkWriteDescriptorSet` structure for each descriptor or descriptor array to update, each of which re-specifies the same state when updating the same descriptor in multiple descriptor sets. For cases when an application wishes to update the same set of descriptors in multiple descriptor sets allocated using the same `VkDescriptorSetLayout`, `vkUpdateDescriptorSetWithTemplate` **can** be used as a replacement for `vkUpdateDescriptorSets`.

`VkDescriptorUpdateTemplate` allows implementations to convert a set of descriptor update operations on a single descriptor set to an internal format that, in conjunction with `vkUpdateDescriptorSetWithTemplate` or `vkCmdPushDescriptorSetWithTemplateKHR`, **can** be more efficient compared to calling `vkUpdateDescriptorSets` or `vkCmdPushDescriptorSetKHR`. The descriptors themselves are not specified in the `VkDescriptorUpdateTemplate`, rather, offsets into an application provided pointer to host memory are specified, which are combined with a pointer passed to `vkUpdateDescriptorSetWithTemplate` or `vkCmdPushDescriptorSetWithTemplateKHR`. This allows large batches of updates to be executed without having to convert application data structures into a strictly-defined Vulkan data structure.

To create a descriptor update template, call:

```
VkResult vkCreateDescriptorUpdateTemplate(  
    VkDevice                                     device,  
    const VkDescriptorUpdateTemplateCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks*                  pAllocator,  
    VkDescriptorUpdateTemplate*                  pDescriptorUpdateTemplate);
```

or the equivalent command

```
VkResult vkCreateDescriptorUpdateTemplateKHR(  
    VkDevice                                     device,  
    const VkDescriptorUpdateTemplateCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks*                  pAllocator,  
    VkDescriptorUpdateTemplate*                  pDescriptorUpdateTemplate);
```

- **device** is the logical device that creates the descriptor update template.
- **pCreateInfo** is a pointer to a `VkDescriptorUpdateTemplateCreateInfo` structure specifying the set of descriptors to update with a single call to `vkCmdPushDescriptorSetWithTemplateKHR` or `vkUpdateDescriptorSetWithTemplate`.
- **pAllocator** controls host memory allocation as described in the [Memory Allocation](#) chapter.
- **pDescriptorUpdateTemplate** is a pointer to a `VkDescriptorUpdateTemplate` handle in which the resulting descriptor update template object is returned.

## Valid Usage (Implicit)

- **device** **must** be a valid `VkDevice` handle
- **pCreateInfo** **must** be a valid pointer to a valid `VkDescriptorUpdateTemplateCreateInfo` structure
- If **pAllocator** is not `NULL`, **pAllocator** **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **pDescriptorUpdateTemplate** **must** be a valid pointer to a `VkDescriptorUpdateTemplate` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDescriptorUpdateTemplateCreateInfo` structure is defined as:

```

typedef struct VkDescriptorUpdateTemplateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorUpdateTemplateCreateFlags flags;
    uint32_t descriptorUpdateEntryCount;
    const VkDescriptorUpdateTemplateEntry* pDescriptorUpdateEntries;
    VkDescriptorUpdateTemplateType templateType;
    VkDescriptorSetLayout descriptorSetLayout;
    VkPipelineBindPoint pipelineBindPoint;
    VkPipelineLayout pipelineLayout;
    uint32_t set;
} VkDescriptorUpdateTemplateCreateInfo;

```

or the equivalent

```
typedef VkDescriptorUpdateTemplateCreateInfo VkDescriptorUpdateTemplateCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **descriptorUpdateEntryCount** is the number of elements in the **pDescriptorUpdateEntries** array.
- **pDescriptorUpdateEntries** is a pointer to an array of **VkDescriptorUpdateTemplateEntry** structures describing the descriptors to be updated by the descriptor update template.
- **templateType** Specifies the type of the descriptor update template. If set to **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_DESCRIPTOR\_SET** it **can** only be used to update descriptor sets with a fixed **descriptorsetLayout**. If set to **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_PUSH\_DESCRIPTORS\_KHR** it **can** only be used to push descriptor sets using the provided **pipelineBindPoint**, **pipelineLayout**, and **set** number.
- **descriptorsetLayout** is the descriptor set layout the parameter update template will be used with. All descriptor sets which are going to be updated through the newly created descriptor update template **must** be created with this layout. **descriptorsetLayout** is the descriptor set layout used to build the descriptor update template. All descriptor sets which are going to be updated through the newly created descriptor update template **must** be created with a layout that matches (is the same as, or defined identically to) this layout. This parameter is ignored if **templateType** is not **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_DESCRIPTOR\_SET**.
- **pipelineBindPoint** is a **VkPipelineBindPoint** indicating whether the descriptors will be used by graphics pipelines or compute pipelines. This parameter is ignored if **templateType** is not **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_PUSH\_DESCRIPTORS\_KHR**
- **pipelineLayout** is a **VkPipelineLayout** object used to program the bindings. This parameter is ignored if **templateType** is not **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_PUSH\_DESCRIPTORS\_KHR**
- **set** is the set number of the descriptor set in the pipeline layout that will be updated. This parameter is ignored if **templateType** is not **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_PUSH\_DESCRIPTORS\_KHR**

## Valid Usage

- If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET`, `descriptorSetLayout` **must** be a valid `VkDescriptorSetLayout` handle
- If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR`, `pipelineBindPoint` **must** be a valid `VkPipelineBindPoint` value
- If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR`, `pipelineLayout` **must** be a valid `VkPipelineLayout` handle
- If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR`, `set` **must** be the unique set number in the pipeline layout that uses a descriptor set layout that was created with `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `pDescriptorUpdateEntries` **must** be a valid pointer to an array of `descriptorUpdateEntryCount` valid `VkDescriptorUpdateTemplateEntry` structures
- `templateType` **must** be a valid `VkDescriptorUpdateTemplateType` value
- `descriptorUpdateEntryCount` **must** be greater than `0`
- Both of `descriptorSetLayout`, and `pipelineLayout` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

```
typedef VkFlags VkDescriptorUpdateTemplateCreateFlags;
```

or the equivalent

```
typedef VkDescriptorUpdateTemplateCreateFlags  
VkDescriptorUpdateTemplateCreateFlagsKHR;
```

`VkDescriptorUpdateTemplateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The descriptor update template type is determined by the `VkDescriptorUpdateTemplateCreateInfo` `::templateType` property, which takes the following values:

```

typedef enum VkDescriptorUpdateTemplateType {
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET = 0,
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR = 1,
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET_KHR =
VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET,
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkDescriptorUpdateTemplateType;

```

or the equivalent

```
typedef VkDescriptorUpdateTemplateType VkDescriptorUpdateTemplateTypeKHR;
```

- **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_DESCRIPTOR\_SET** specifies that the descriptor update template will be used for descriptor set updates only.
- **VK\_DESCRIPTOR\_UPDATE\_TEMPLATE\_TYPE\_PUSH\_DESCRIPTORS\_KHR** specifies that the descriptor update template will be used for push descriptor updates only.

The **VkDescriptorUpdateTemplateEntry** structure is defined as:

```

typedef struct VkDescriptorUpdateTemplateEntry {
    uint32_t          dstBinding;
    uint32_t          dstArrayElement;
    uint32_t          descriptorCount;
    VkDescriptorType  descriptorType;
    size_t            offset;
    size_t            stride;
} VkDescriptorUpdateTemplateEntry;

```

or the equivalent

```
typedef VkDescriptorUpdateTemplateEntry VkDescriptorUpdateTemplateEntryKHR;
```

- **dstBinding** is the descriptor binding to update when using this descriptor update template.
- **dstArrayElement** is the starting element in the array belonging to **dstBinding**. If the descriptor binding identified by **srcBinding** has a descriptor type of **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT** then **dstArrayElement** specifies the starting byte offset to update.
- **descriptorCount** is the number of descriptors to update. If **descriptorCount** is greater than the number of remaining array elements in the destination binding, those affect consecutive bindings in a manner similar to **VkWriteDescriptorSet** above. If the descriptor binding identified by **dstBinding** has a descriptor type of **VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT** then **descriptorCount** specifies the number of bytes to update and the remaining array elements in the destination binding refer to the remaining number of bytes in it.
- **descriptorType** is a **VkDescriptorType** specifying the type of the descriptor.

- **offset** is the offset in bytes of the first binding in the raw data structure.
- **stride** is the stride in bytes between two consecutive array elements of the descriptor update informations in the raw data structure. The actual pointer ptr for each array element j of update entry i is computed using the following formula:

```
const char *ptr = (const char *)pData + pDescriptorUpdateEntries[i].offset + j
* pDescriptorUpdateEntries[i].stride
```

The stride is useful in case the bindings are stored in structs along with other data. If **descriptorType** is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then the value of **stride** is ignored and the stride is assumed to be `1`, i.e. the descriptor update information for them is always specified as a contiguous range.

## Valid Usage

- **dstBinding** **must** be a valid binding in the descriptor set layout implicitly specified when using a descriptor update template to update descriptors.
- **dstArrayElement** and **descriptorCount** **must** be less than or equal to the number of array elements in the descriptor set binding implicitly specified when using a descriptor update template to update descriptors, and all applicable consecutive bindings, as described by [consecutive binding updates](#)
- If **descriptor** type is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, **dstArrayElement** **must** be an integer multiple of `4`
- If **descriptor** type is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, **descriptorCount** **must** be an integer multiple of `4`

## Valid Usage (Implicit)

- **descriptorType** **must** be a valid `VkDescriptorType` value

To destroy a descriptor update template, call:

```
void vkDestroyDescriptorUpdateTemplate(
    VkDevice                                     device,
    VkDescriptorUpdateTemplate                  descriptorUpdateTemplate,
    const VkAllocationCallbacks* pAllocator);
```

or the equivalent command

```
void vkDestroyDescriptorUpdateTemplateKHR(  
    VkDevice device,  
    VkDescriptorUpdateTemplate descriptorUpdateTemplate,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that has been used to create the descriptor update template
- `descriptorUpdateTemplate` is the descriptor update template to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- If `VkAllocationCallbacks` were provided when `descriptorsetLayout` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `descriptorsetLayout` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `descriptorUpdateTemplate` is not `VK_NULL_HANDLE`, `descriptorUpdateTemplate` **must** be a valid `VkDescriptorUpdateTemplate` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `descriptorUpdateTemplate` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `descriptorUpdateTemplate` **must** be externally synchronized

Once a `VkDescriptorUpdateTemplate` has been created, descriptor sets **can** be updated by calling:

```
void vkUpdateDescriptorSetWithTemplate(  
    VkDevice device,  
    VkDescriptorSet descriptorSet,  
    VkDescriptorUpdateTemplate descriptorUpdateTemplate,  
    const void* pData);
```

or the equivalent command

```
void vkUpdateDescriptorSetWithTemplateKHR(
    VkDevice                                     device,
    VkDescriptorSet                             descriptorSet,
    VkDescriptorUpdateTemplate                 descriptorUpdateTemplate,
    const void*                                 pData);
```

- `device` is the logical device that updates the descriptor sets.
- `descriptorSet` is the descriptor set to update
- `descriptorUpdateTemplate` is a `VkDescriptorUpdateTemplate` object specifying the update mapping between `pData` and the descriptor set to update.
- `pData` is a pointer to memory containing one or more `VkDescriptorImageInfo`, `VkDescriptorBufferInfo`, or `VkBufferView` structures used to write the descriptors.

## Valid Usage

- `pData` **must** be a valid pointer to a memory containing one or more valid instances of `VkDescriptorImageInfo`, `VkDescriptorBufferInfo`, or `VkBufferView` in a layout defined by `descriptorUpdateTemplate` when it was created with `vkCreateDescriptorUpdateTemplate`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `descriptorSet` **must** be a valid `VkDescriptorSet` handle
- `descriptorUpdateTemplate` **must** be a valid `VkDescriptorUpdateTemplate` handle
- `descriptorUpdateTemplate` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `descriptorSet` **must** be externally synchronized

*API example*

```
struct AppBufferView {
    VkBufferView bufferView;
    uint32_t     applicationRelatedInformation;
};

struct AppDataStructure
{
    VkDescriptorImageInfo   imageInfo;           // a single image info
    VkDescriptorBufferInfo bufferInfoArray[3]; // 3 buffer infos in an array
    AppBufferView          bufferView[2];        // An application defined structure
```

```

containing a bufferView
    // ... some more application related data
};

const VkDescriptorUpdateTemplateEntry descriptorUpdateTemplateEntries[] =
{
    // binding to a single image descriptor
    {
        0,                                // binding
        0,                                // dstArrayElement
        1,                                // descriptorCount
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, // descriptorType
        offsetof(AppDataStructure, imageInfo),   // offset
        0                                   // stride is not required if
descriptorCount is 1
    },

    // binding to an array of buffer descriptors
    {
        1,                                // binding
        0,                                // dstArrayElement
        3,                                // descriptorCount
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, // descriptorType
        offsetof(AppDataStructure, bufferInfoArray), // offset
        sizeof(VkDescriptorBufferInfo)           // stride, descriptor buffer
infos are compact
    },

    // binding to an array of buffer views
    {
        2,                                // binding
        0,                                // dstArrayElement
        2,                                // descriptorCount
        VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, // descriptorType
        offsetof(AppDataStructure, bufferView) + // offset
            offsetof(AppBufferView, bufferView), // stride
        sizeof(AppBufferView)                // stride, bufferViews do not
have to be compact
    },
};

// create a descriptor update template for descriptor set updates
const VkDescriptorUpdateTemplateCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO, // sType
    NULL,                                         // pNext
    0,                                            // flags
    3,                                            // /
descriptorUpdateEntryCount
    descriptorUpdateTemplateEntries,               // /
pDescriptorUpdateEntries
};

```

```

VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET,           // templateType
myLayout,                                                 // descriptorSetLayout
0,                                                       // pipelineBindPoint,
ignored by given templateType
0,                                                       // pipelineLayout,
ignored by given templateType
0,                                                       // set, ignored by
given templateType
};

VkDescriptorUpdateTemplate myDescriptorUpdateTemplate;
myResult = vkCreateDescriptorUpdateTemplate(
    myDevice,
    &CreateInfo,
    NULL,
    &myDescriptorUpdateTemplate);
}

```

AppDataStructure appData;

```

// fill appData here or cache it in your engine
vkUpdateDescriptorSetWithTemplate(myDevice, myDescriptorSet,
myDescriptorUpdateTemplate, &appData);

```

### 13.2.7. Descriptor Set Binding

To bind one or more descriptor sets to a command buffer, call:

```

void vkCmdBindDescriptorSets(
    VkCommandBuffer
    VkPipelineBindPoint
    VkPipelineLayout
    uint32_t
    uint32_t
    const VkDescriptorSet*
    uint32_t
    const uint32_t*
    commandBuffer,
    pipelineBindPoint,
    layout,
    firstSet,
    descriptorSetCount,
    pDescriptorSets,
    dynamicOffsetCount,
    pDynamicOffsets);

```

- **commandBuffer** is the command buffer that the descriptor sets will be bound to.
- **pipelineBindPoint** is a [VkPipelineBindPoint](#) indicating whether the descriptors will be used by graphics pipelines or compute pipelines. There is a separate set of bind points for each of graphics and compute, so binding one does not disturb the other.
- **layout** is a [VkPipelineLayout](#) object used to program the bindings.
- **firstSet** is the set number of the first descriptor set to be bound.
- **descriptorSetCount** is the number of elements in the **pDescriptorSets** array.
- **pDescriptorSets** is a pointer to an array of handles to [VkDescriptorSet](#) objects describing the

descriptor sets to write to.

- `dynamicOffsetCount` is the number of dynamic offsets in the `pDynamicOffsets` array.
- `pDynamicOffsets` is a pointer to an array of `uint32_t` values specifying dynamic offsets.

`vkCmdBindDescriptorSets` causes the sets numbered [`firstSet.. firstSet+descriptorSetCount-1`] to use the bindings stored in `pDescriptorSets[0..descriptorSetCount-1]` for subsequent rendering commands (either compute or graphics, according to the `pipelineBindPoint`). Any bindings that were previously applied via these sets are no longer valid.

Once bound, a descriptor set affects rendering of subsequent graphics or compute commands in the command buffer until a different set is bound to the same set number, or else until the set is disturbed as described in [Pipeline Layout Compatibility](#).

A compatible descriptor set **must** be bound for all set numbers that any shaders in a pipeline access, at the time that a draw or dispatch command is recorded to execute using that pipeline. However, if none of the shaders in a pipeline statically use any bindings with a particular set number, then no descriptor set need be bound for that set number, even if the pipeline layout includes a non-trivial descriptor set layout for that set number.

If any of the sets being bound include dynamic uniform or storage buffers, then `pDynamicOffsets` includes one element for each array element in each dynamic descriptor type binding in each set. Values are taken from `pDynamicOffsets` in an order such that all entries for set N come before set N+1; within a set, entries are ordered by the binding numbers in the descriptor set layouts; and within a binding array, elements are in order. `dynamicOffsetCount` **must** equal the total number of dynamic descriptors in the sets being bound.

The effective offset used for dynamic uniform and storage buffer bindings is the sum of the relative offset taken from `pDynamicOffsets`, and the base address of the buffer plus base offset in the descriptor set. The range of the dynamic uniform and storage buffer bindings is the buffer range as specified in the descriptor set.

Each of the `pDescriptorSets` **must** be compatible with the pipeline layout specified by `layout`. The layout used to program the bindings **must** also be compatible with the pipeline used in subsequent graphics or compute commands, as defined in the [Pipeline Layout Compatibility](#) section.

The descriptor set contents bound by a call to `vkCmdBindDescriptorSets` **may** be consumed at the following times:

- For descriptor bindings created with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` bit set, the contents **may** be consumed when the command buffer is submitted to a queue, or during shader execution of the resulting draws and dispatches, or any time in between. Otherwise,
- during host execution of the command, or during shader execution of the resulting draws and dispatches, or any time in between.

Thus, the contents of a descriptor set binding **must** not be altered (overwritten by an update command, or freed) between the first point in time that it **may** be consumed, and when the command completes executing on the queue.

The contents of `pDynamicOffsets` are consumed immediately during execution of `vkCmdBindDescriptorSets`. Once all pending uses have completed, it is legal to update and reuse a descriptor set.

## Valid Usage

- Each element of `pDescriptorSets` **must** have been allocated with a `VkDescriptorSetLayout` that matches (is the same as, or identically defined as) the `VkDescriptorSetLayout` at set  $n$  in `layout`, where  $n$  is the sum of `firstSet` and the index into `pDescriptorSets`
- `dynamicOffsetCount` **must** be equal to the total number of dynamic descriptors in `pDescriptorSets`
- The sum of `firstSet` and `descriptorSetCount` **must** be less than or equal to `VkPipelineLayoutCreateInfo::setLayoutCount` provided when `layout` was created
- `pipelineBindPoint` **must** be supported by the `commandBuffer`'s parent `VkCommandPool`'s queue family
- Each element of `pDynamicOffsets` which corresponds to a descriptor binding with type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` **must** be a multiple of `VkPhysicalDeviceLimits ::minUniformBufferOffsetAlignment`
- Each element of `pDynamicOffsets` which corresponds to a descriptor binding with type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` **must** be a multiple of `VkPhysicalDeviceLimits ::minStorageBufferOffsetAlignment`
- For each dynamic uniform or storage buffer binding in `pDescriptorSets`, the sum of the effective offset, as defined above, and the range of the binding **must** be less than or equal to the size of the buffer

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pipelineBindPoint` **must** be a valid `VkPipelineBindPoint` value
- `layout` **must** be a valid `VkPipelineLayout` handle
- `pDescriptorSets` **must** be a valid pointer to an array of `descriptorSetCount` valid `VkDescriptorSet` handles
- If `dynamicOffsetCount` is not `0`, `pDynamicOffsets` **must** be a valid pointer to an array of `dynamicOffsetCount uint32_t` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- `descriptorSetCount` **must** be greater than `0`
- Each of `commandBuffer`, `layout`, and the elements of `pDescriptorSets` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

### 13.2.8. Push Descriptor Updates

In addition to allocating descriptor sets and binding them to a command buffer, an application **can** record descriptor updates into the command buffer.

To push descriptor updates into a command buffer, call:

```
void vkCmdPushDescriptorSetKHR(  
    VkCommandBuffer  
    VkPipelineBindPoint  
    VkPipelineLayout  
    uint32_t  
    uint32_t  
    const VkWriteDescriptorSet*  
                                commandBuffer,  
                                pipelineBindPoint,  
                                layout,  
                                set,  
                                descriptorWriteCount,  
                                pDescriptorWrites);
```

- `commandBuffer` is the command buffer that the descriptors will be recorded in.
- `pipelineBindPoint` is a `VkPipelineBindPoint` indicating whether the descriptors will be used by graphics pipelines or compute pipelines. There is a separate set of push descriptor bindings for each of graphics and compute, so binding one does not disturb the other.
- `layout` is a `VkPipelineLayout` object used to program the bindings.
- `set` is the set number of the descriptor set in the pipeline layout that will be updated.
- `descriptorWriteCount` is the number of elements in the `pDescriptorWrites` array.
- `pDescriptorWrites` is a pointer to an array of `VkWriteDescriptorSet` structures describing the descriptors to be updated.

*Push descriptors* are a small bank of descriptors whose storage is internally managed by the command buffer rather than being written into a descriptor set and later bound to a command buffer. Push descriptors allow for incremental updates of descriptors without managing the lifetime of descriptor sets.

When a command buffer begins recording, all push descriptors are undefined. Push descriptors **can** be updated incrementally and cause shaders to use the updated descriptors for subsequent rendering commands (either compute or graphics, according to the `pipelineBindPoint`) until the descriptor is overwritten, or else until the set is disturbed as described in [Pipeline Layout Compatibility](#). When the set is disturbed or push descriptors with a different descriptor set layout are set, all push descriptors are undefined.

Push descriptors that are [statically used](#) by a pipeline **must** not be undefined at the time that a draw or dispatch command is recorded to execute using that pipeline. This includes immutable sampler descriptors, which **must** be pushed before they are accessed by a pipeline (the immutable samplers are pushed, rather than the samplers in `pDescriptorWrites`). Push descriptors that are not statically used **can** remain undefined.

Push descriptors do not use dynamic offsets. Instead, the corresponding non-dynamic descriptor types **can** be used and the `offset` member of `VkDescriptorBufferInfo` **can** be changed each time the descriptor is written.

Each element of `pDescriptorWrites` is interpreted as in `VkWriteDescriptorSet`, except the `dstSet` member is ignored.

To push an immutable sampler, use a `VkWriteDescriptorSet` with `dstBinding` and `dstArrayElement` selecting the immutable sampler's binding. If the descriptor type is `VK_DESCRIPTOR_TYPE_SAMPLER`, the `pImageInfo` parameter is ignored and the immutable sampler is taken from the push descriptor set layout in the pipeline layout. If the descriptor type is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `sampler` member of the `pImageInfo` parameter is ignored and the immutable sampler is taken from the push descriptor set layout in the pipeline layout.

## Valid Usage

- `pipelineBindPoint` **must** be supported by the `commandBuffer`'s parent `VkCommandPool`'s queue family
- `set` **must** be less than `VkPipelineLayoutCreateInfo::setLayoutCount` provided when `layout` was created
- `set` **must** be the unique set number in the pipeline layout that uses a descriptor set layout that was created with `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pipelineBindPoint` **must** be a valid `VkPipelineBindPoint` value
- `layout` **must** be a valid `VkPipelineLayout` handle
- `pDescriptorWrites` **must** be a valid pointer to an array of `descriptorWriteCount` valid `VkWriteDescriptorSet` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- `descriptorWriteCount` **must** be greater than `0`
- Both of `commandBuffer`, and `layout` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

### 13.2.9. Push Descriptor Updates with Descriptor Update Templates

It is also possible to use a descriptor update template to specify the push descriptors to update. To do so, call:

```
void vkCmdPushDescriptorSetWithTemplateKHR(  
    VkCommandBuffer  
    VkDescriptorUpdateTemplate  
    VkPipelineLayout  
    uint32_t  
    const void*
```

```
        commandBuffer,  
        descriptorUpdateTemplate,  
        layout,  
        set,  
        pData);
```

- `commandBuffer` is the command buffer that the descriptors will be recorded in.

- `descriptorUpdateTemplate` is a descriptor update template defining how to interpret the descriptor information in `pData`.
- `layout` is a `VkPipelineLayout` object used to program the bindings. It **must** be compatible with the layout used to create the `descriptorUpdateTemplate` handle.
- `set` is the set number of the descriptor set in the pipeline layout that will be updated. This **must** be the same number used to create the `descriptorUpdateTemplate` handle.
- `pData` is a pointer to memory containing descriptors for the templated update.

## Valid Usage

- The `pipelineBindPoint` specified during the creation of the descriptor update template **must** be supported by the `commandBuffer`'s parent `VkCommandPool`'s queue family
- `pData` **must** be a valid pointer to a memory containing one or more valid instances of `VkDescriptorImageInfo`, `VkDescriptorBufferInfo`, or `VkBufferView` in a layout defined by `descriptorUpdateTemplate` when it was created with `vkCreateDescriptorUpdateTemplateKHR`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `descriptorUpdateTemplate` **must** be a valid `VkDescriptorUpdateTemplate` handle
- `layout` **must** be a valid `VkPipelineLayout` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- Each of `commandBuffer`, `descriptorUpdateTemplate`, and `layout` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

*API example*

```

struct AppDataStructure
{
    VkDescriptorImageInfo imageInfo;           // a single image info
    // ... some more application related data
};

const VkDescriptorUpdateTemplateEntry descriptorUpdateTemplateEntries[] =
{
    // binding to a single image descriptor
    {
        0,                                // binding
        0,                                // dstArrayElement
        1,                                // descriptorCount
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, // descriptorType
        offsetof(AppDataStructure, imageInfo),   // offset
        0                                   // stride is not required if
descriptorCount is 1
    }
};

// create a descriptor update template for descriptor set updates
const VkDescriptorUpdateTemplateCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO, // sType
    NULL,                                         // pNext
    0,                                            // flags
    1,                                            // 
descriptorUpdateEntryCount
    descriptorUpdateTemplateEntries,                // 
DescriptorUpdateEntries
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR, // templateType
    0,                                            // descriptorSetLayout,
ignored by given templateType
    VK_PIPELINE_BIND_POINT_GRAPHICS,               // pipelineBindPoint
    myPipelineLayout,                            // pipelineLayout
    0,                                            // set
};

VkDescriptorUpdateTemplate myDescriptorUpdateTemplate;
myResult = vkCreateDescriptorUpdateTemplate(

```

```

    myDevice,
    &createInfo,
    NULL,
    &myDescriptorUpdateTemplate);
}

AppDataStructure appData;
// fill appData here or cache it in your engine
vkCmdPushDescriptorSetWithTemplateKHR(myCmdBuffer, myDescriptorUpdateTemplate,
myPipelineLayout, 0, &appData);

```

### 13.2.10. Push Constant Updates

As described above in section [Pipeline Layouts](#), the pipeline layout defines shader push constants which are updated via Vulkan commands rather than via writes to memory or copy commands.

*Note*



Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

The values of push constants are undefined at the start of a command buffer.

To update push constants, call:

```

void vkCmdPushConstants(
    VkCommandBuffer           commandBuffer,
    VkPipelineLayout          layout,
    VkShaderStageFlags        stageFlags,
    uint32_t                  offset,
    uint32_t                  size,
    const void*               pValues);

```

- `commandBuffer` is the command buffer in which the push constant update will be recorded.
- `layout` is the pipeline layout used to program the push constant updates.
- `stageFlags` is a bitmask of [VkShaderStageFlagBits](#) specifying the shader stages that will use the push constants in the updated range.
- `offset` is the start offset of the push constant range to update, in units of bytes.
- `size` is the size of the push constant range to update, in units of bytes.
- `pValues` is a pointer to an array of `size` bytes containing the new push constant values.

*Note*



As `stageFlags` needs to include all flags the relevant push constant ranges were created with, any flags that are not supported by the queue family that the [VkCommandPool](#) used to allocate `commandBuffer` was created on are ignored.

## Valid Usage

- For each byte in the range specified by `offset` and `size` and for each shader stage in `stageFlags`, there **must** be a push constant range in `layout` that includes that byte and that stage
- For each byte in the range specified by `offset` and `size` and for each push constant range that overlaps that byte, `stageFlags` **must** include all stages in that push constant range's `VkPushConstantRange::stageFlags`
- `offset` **must** be a multiple of 4
- `size` **must** be a multiple of 4
- `offset` **must** be less than `VkPhysicalDeviceLimits::maxPushConstantsSize`
- `size` **must** be less than or equal to `VkPhysicalDeviceLimits::maxPushConstantsSize` minus `offset`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `layout` **must** be a valid `VkPipelineLayout` handle
- `stageFlags` **must** be a valid combination of `VkShaderStageFlagBits` values
- `stageFlags` **must** not be 0
- `pValues` **must** be a valid pointer to an array of `size` bytes
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- `size` **must** be greater than 0
- Both of `commandBuffer`, and `layout` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

### 13.3. Physical Storage Buffer Access

To query a 64-bit buffer device address value through which buffer memory **can** be accessed in a shader, call:

```
VkDeviceAddress vkGetBufferDeviceAddressKHR(  
    VkDevice device,  
    const VkBufferDeviceAddressInfoKHR* pInfo);
```

or the equivalent command

```
VkDeviceAddress vkGetBufferDeviceAddressEXT(  
    VkDevice device,  
    const VkBufferDeviceAddressInfoKHR* pInfo);
```

- **device** is the logical device that the buffer was created on.
- **pInfo** is a pointer to a `VkBufferDeviceAddressInfoKHR` structure specifying the buffer to retrieve an address for.

The 64-bit return value is an address of the start of **pInfo::buffer**. The address range starting at this value and whose size is the size of the buffer **can** be used in a shader to access the memory bound to that buffer, using the `SPV_KHR_physical_storage_buffer` extension or the equivalent `SPV_EXT_physical_storage_buffer` extension and the `PhysicalStorageBuffer` storage class. For example, this value **can** be stored in a uniform buffer, and the shader **can** read the value from the uniform buffer and use it to do a dependent read/write to this buffer. A value of zero is reserved as a “null” pointer and **must** not be returned as a valid buffer device address. All loads, stores, and atomics in a shader through `PhysicalStorageBuffer` pointers **must** access addresses in the address range of some buffer.

If the buffer was created with a non-zero value of `VkBufferOpaqueCaptureAddressCreateInfoKHR::opaqueCaptureAddress` or `VkBufferDeviceAddressCreateInfoEXT::deviceAddress` the return value will be the same address that was returned at capture time.

## Valid Usage

- The `bufferDeviceAddress` or `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT ::bufferDeviceAddress` feature **must** be enabled
- If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` or `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT ::bufferDeviceAddressMultiDevice` feature **must** be enabled

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkBufferDeviceAddressInfoKHR` structure

The `VkBufferDeviceAddressInfoKHR` structure is defined as:

```
typedef struct VkBufferDeviceAddressInfoKHR {  
    VkStructureType    sType;  
    const void*        pNext;  
    VkBuffer          buffer;  
} VkBufferDeviceAddressInfoKHR;
```

or the equivalent

```
typedef VkBufferDeviceAddressInfoKHR VkBufferDeviceAddressInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `buffer` specifies the buffer whose address is being queried.

## Valid Usage

- If `buffer` is non-sparse and was not created with the `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR` flag, then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR`
- `pNext` **must** be `NULL`
- `buffer` **must** be a valid `VkBuffer` handle

To query a 64-bit buffer opaque capture address, call:

```
uint64_t vkGetBufferOpaqueCaptureAddressKHR(  
    VkDevice device,  
    const VkBufferDeviceAddressInfoKHR* pInfo);
```

- `device` is the logical device that the buffer was created on.
- `pInfo` is a pointer to an instance of the `VkBufferDeviceAddressInfoKHR` structure specifying the buffer to retrieve an address for.

The 64-bit return value is an opaque capture address of the start of `pInfo::buffer`.

If the buffer was created with a non-zero value of `VkBufferOpaqueCaptureAddressCreateInfoKHR` `::opaqueCaptureAddress` the return value **must** be the same address.

## Valid Usage

- The `bufferDeviceAddress` feature **must** be enabled
- If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature **must** be enabled

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkBufferDeviceAddressInfoKHR` structure

# Chapter 14. Shader Interfaces

When a pipeline is created, the set of shaders specified in the corresponding `Vk*PipelineCreateInfo` structure are implicitly linked at a number of different interfaces.

- [Shader Input and Output Interface](#)
- [Vertex Input Interface](#)
- [Fragment Output Interface](#)
- [Fragment Input Attachment Interface](#)
- [Shader Resource Interface](#)

Interface definitions make use of the following SPIR-V decorations:

- [DescriptorSet](#) and [Binding](#)
- [Location](#), [Component](#), and [Index](#)
- [Flat](#), [NoPerspective](#), [Centroid](#), and [Sample](#)
- [Block](#) and [BufferBlock](#)
- [InputAttachmentIndex](#)
- [Offset](#), [ArrayStride](#), and [MatrixStride](#)
- [BuiltIn](#)
- [PassthroughNV](#)

This specification describes valid uses for Vulkan of these decorations. Any other use of one of these decorations is invalid.

## 14.1. Shader Input and Output Interfaces

When multiple stages are present in a pipeline, the outputs of one stage form an interface with the inputs of the next stage. When such an interface involves a shader, shader outputs are matched against the inputs of the next stage, and shader inputs are matched against the outputs of the previous stage.

There are two classes of variables that **can** be matched between shader stages, built-in variables and user-defined variables. Each class has a different set of matching criteria. Generally, when non-shader stages are between shader stages, the user-defined variables, and most built-in variables, form an interface between the shader stages.

The variables forming the input or output *interfaces* are listed as operands to the `OpEntryPoint` instruction and are declared with the [Input](#) or [Output](#) storage classes, respectively, in the SPIR-V module.

[Output](#) variables of a shader stage have undefined values until the shader writes to them or uses the [Initializer](#) operand when declaring the variable.

### 14.1.1. Built-in Interface Block

Shader [built-in](#) variables meeting the following requirements define the *built-in interface block*. They **must**

- be explicitly declared (there are no implicit built-ins),
- be identified with a [BuiltIn](#) decoration,
- form object types as described in the [Built-in Variables](#) section, and
- be declared in a block whose top-level members are the built-ins.

Built-ins only participate in interface matching if they are declared in such a block. They **must** not have any [Location](#) or [Component](#) decorations.

There **must** be no more than one built-in interface block per shader per interface.

### 14.1.2. User-defined Variable Interface

The remaining variables listed by [OpEntryPoint](#) with the [Input](#) or [Output](#) storage class form the *user-defined variable interface*. These **must** have SPIR-V numerical types or, recursively, composite types of such types. By default, the components of such types have a width of 32 or 64 bits. If an implementation supports [storageInputOutput16](#), components **can** also have a width of 16 bits. These variables **must** be identified with a [Location](#) decoration and **can** also be identified with a [Component](#) decoration.

### 14.1.3. Interface Matching

A user-defined output variable is considered to match an input variable in the subsequent stage if the two variables are declared with the same [Location](#) and [Component](#) decoration and match in type and decoration, except that [interpolation decorations](#) are not **required** to match. [XfbBuffer](#), [XfbStride](#), [Offset](#), and [Stream](#) are also not required to match for the purposes of interface matching. For the purposes of interface matching, variables declared without a [Component](#) decoration are considered to have a [Component](#) decoration of zero.

*Note*



Matching rules for *passthrough geometry shaders* are slightly different and are described in the [Passthrough Interface Matching](#) section.

Variables or block members declared as structures are considered to match in type if and only if the structure members match in type, decoration, number, and declaration order. Variables or block members declared as arrays are considered to match in type only if both declarations specify the same element type and size.

Tessellation control and mesh shader per-vertex output variables and blocks, and tessellation control, tessellation evaluation, and geometry shader per-vertex input variables and blocks are required to be declared as arrays, with each element representing input or output values for a single vertex of a multi-vertex primitive. For the purposes of interface matching, the outermost array dimension of such variables and blocks is ignored.

At an interface between two non-fragment shader stages, the built-in interface block **must** match exactly, as described above, except for per-view outputs as described in [Mesh Shader Per-View Outputs](#). At an interface involving the fragment shader inputs, the presence or absence of any built-in output does not affect the interface matching.

At an interface between two shader stages, the user-defined variable interface **must** match exactly, as described above.

Any input value to a shader stage is well-defined as long as the preceding stages writes to a matching output, as described above.

Additionally, scalar and vector inputs are well-defined if there is a corresponding output satisfying all of the following conditions:

- the input and output match exactly in decoration,
- the output is a vector with the same basic type and has at least as many components as the input, and
- the common component type of the input and output is 16-bit integer or floating-point, or 32-bit integer or floating-point (64-bit component types are excluded).

In this case, the components of the input will be taken from the first components of the output, and any extra components of the output will be ignored.

#### 14.1.4. Location Assignment

This section describes how many locations are consumed by a given type. As mentioned above, geometry shader inputs, tessellation control shader inputs and outputs, and tessellation evaluation inputs all have an additional level of arrayness relative to other shader inputs and outputs. This outer array level is removed from the type before considering how many locations the type consumes.

The **Location** value specifies an interface slot comprised of a 32-bit four-component vector conveyed between stages. The **Component** specifies **components** within these vector locations. Only types with widths of 16, 32 or 64 are supported in shader interfaces.

Inputs and outputs of the following types consume a single interface location:

- 16-bit scalar and vector types, and
- 32-bit scalar and vector types, and
- 64-bit scalar and 2-component vector types.

64-bit three- and four-component vectors consume two consecutive locations.

If a declared input or output is an array of size  $n$  and each element takes  $m$  locations, it will be assigned  $m \times n$  consecutive locations starting with the location specified.

If the declared input or output is an  $n \times m$  16-, 32- or 64-bit matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an  $n$ -element array of  $m$ -component vectors.

The layout of a structure type used as an [Input](#) or [Output](#) depends on whether it is also a [Block](#) (i.e. has a [Block](#) decoration).

If it is not a [Block](#), then the structure type **must** have a [Location](#) decoration. Its members are assigned consecutive locations in their declaration order, with the first member assigned to the location specified for the structure type. The members, and their nested types, **must** not themselves have [Location](#) decorations.

If the structure type is a [Block](#) but without a [Location](#), then each of its members **must** have a [Location](#) decoration. If it is a [Block](#) with a [Location](#) decoration, then its members are assigned consecutive locations in declaration order, starting from the first member which is initially assigned the location specified for the [Block](#). Any member with its own [Location](#) decoration is assigned that location. Each remaining member is assigned the location after the immediately preceding member in declaration order.

The locations consumed by block and structure members are determined by applying the rules above in a depth-first traversal of the instantiated members as though the structure or block member were declared as an input or output variable of the same type.

Any two inputs listed as operands on the same [OpEntryPoint](#) **must** not be assigned the same location, either explicitly or implicitly. Any two outputs listed as operands on the same [OpEntryPoint](#) **must** not be assigned the same location, either explicitly or implicitly.

The number of input and output locations available for a shader input or output interface are limited, and dependent on the shader stage as described in [Shader Input and Output Locations](#). All variables in both the [built-in interface block](#) and the [user-defined variable interface](#) count against these limits. Each effective [Location](#) **must** have a value less than the number of locations available for the given interface, as specified in the "Locations Available" column in [Shader Input and Output Locations](#).

*Table 18. Shader Input and Output Locations*

Shader Interface	Locations Available
vertex input	<code>maxVertexInputAttributes</code>
vertex output	<code>maxVertexOutputComponents / 4</code>
tessellation control input	<code>maxTessellationControlPerVertexInputComponents / 4</code>
tessellation control output	<code>maxTessellationControlPerVertexOutputComponents / 4</code>
tessellation evaluation input	<code>maxTessellationEvaluationInputComponents / 4</code>
tessellation evaluation output	<code>maxTessellationEvaluationOutputComponents / 4</code>
geometry input	<code>maxGeometryInputComponents / 4</code>
geometry output	<code>maxGeometryOutputComponents / 4</code>
fragment input	<code>maxFragmentInputComponents / 4</code>
fragment output	<code>maxFragmentOutputAttachments</code>

### 14.1.5. Component Assignment

The [Component](#) decoration allows the [Location](#) to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable or block member starting at component N will consume components N, N+1, N+2, ... up through its size. For 16-, and 32-bit types, it is invalid if this sequence of components gets larger than 3. A scalar 64-bit type will consume two of these components in sequence, and a two-component 64-bit vector type will consume all four components available within a location. A three- or four-component 64-bit vector type **must** not specify a [Component](#) decoration. A three-component 64-bit vector type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations.

A scalar or two-component 64-bit data type **must** not specify a [Component](#) decoration of 1 or 3. A [Component](#) decoration **must** not be specified for any type that is not a scalar or vector.

## 14.2. Vertex Input Interface

When the vertex stage is present in a pipeline, the vertex shader input variables form an interface with the vertex input attributes. The vertex shader input variables are matched by the [Location](#) and [Component](#) decorations to the vertex input attributes specified in the [pVertexInputState](#) member of the [VkGraphicsPipelineCreateInfo](#) structure.

The vertex shader input variables listed by [OpEntryPoint](#) with the [Input](#) storage class form the *vertex input interface*. These variables **must** be identified with a [Location](#) decoration and **can** also be identified with a [Component](#) decoration.

For the purposes of interface matching: variables declared without a [Component](#) decoration are considered to have a [Component](#) decoration of zero. The number of available vertex input locations is given by the [maxVertexInputAttributes](#) member of the [VkPhysicalDeviceLimits](#) structure.

See [Attribute Location and Component Assignment](#) for details.

All vertex shader inputs declared as above **must** have a corresponding attribute and binding in the pipeline.

## 14.3. Fragment Output Interface

When the fragment stage is present in a pipeline, the fragment shader outputs form an interface with the output attachments of the current subpass. The fragment shader output variables are matched by the [Location](#) and [Component](#) decorations to the color attachments specified in the [pColorAttachments](#) array of the [VkSubpassDescription](#) structure describing the subpass that the fragment shader is executed in.

The fragment shader output variables listed by [OpEntryPoint](#) with the [Output](#) storage class form the *fragment output interface*. These variables **must** be identified with a [Location](#) decoration. They **can** also be identified with a [Component](#) decoration and/or an [Index](#) decoration. For the purposes of interface matching: variables declared without a [Component](#) decoration are considered to have a [Component](#) decoration of zero, and variables declared without an [Index](#) decoration are considered to

have an [Index](#) decoration of zero.

A fragment shader output variable identified with a [Location](#) decoration of  $i$  is directed to the color attachment indicated by `pColorAttachments[i]`, after passing through the blending unit as described in [Blending](#), if enabled. Locations are consumed as described in [Location Assignment](#). The number of available fragment output locations is given by the `maxFragmentOutputAttachments` member of the `VkPhysicalDeviceLimits` structure.

Components of the output variables are assigned as described in [Component Assignment](#). Output components identified as 0, 1, 2, and 3 will be directed to the R, G, B, and A inputs to the blending unit, respectively, or to the output attachment if blending is disabled. If two variables are placed within the same location, they **must** have the same underlying type (floating-point or integer). The input values to blending or color attachment writes are undefined for components which do not correspond to a fragment shader output.

Fragment outputs identified with an [Index](#) of zero are directed to the first input of the blending unit associated with the corresponding [Location](#). Outputs identified with an [Index](#) of one are directed to the second input of the corresponding blending unit.

No *component aliasing* of output variables is allowed, that is there **must** not be two output variables which have the same location, component, and index, either explicitly declared or implied.

Output values written by a fragment shader **must** be declared with either `OpTypeFloat` or `OpTypeInt`, and a `Width` of 32. If `storageInputOutput16` is supported, output values written by a fragment shader **can** be also declared with either `OpTypeFloat` or `OpTypeInt` and a `Width` of 16. Composites of these types are also permitted. If the color attachment has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in [Conversion from Floating-Point to Normalized Fixed-Point](#); If the color attachment has an integer format, color values are assumed to be integers and converted to the bit-depth of the target. Any value that cannot be represented in the attachment's format is undefined. For any other attachment format no conversion is performed. If the type of the values written by the fragment shader do not match the format of the corresponding color attachment, the resulting values are undefined for those components.

## 14.4. Fragment Input Attachment Interface

When a fragment stage is present in a pipeline, the fragment shader subpass inputs form an interface with the input attachments of the current subpass. The fragment shader subpass input variables are matched by `InputAttachmentIndex` decorations to the input attachments specified in the `pInputAttachments` array of the `VkSubpassDescription` structure describing the subpass that the fragment shader is executed in.

The fragment shader subpass input variables with the `UniformConstant` storage class and a decoration of `InputAttachmentIndex` that are statically used by `OpEntryPoint` form the *fragment input attachment interface*. These variables **must** be declared with a type of `OpTypeImage`, a `Dim` operand of `SubpassData`, and a `Sampled` operand of 2.

A subpass input variable identified with an `InputAttachmentIndex` decoration of  $i$  reads from the input attachment indicated by `pInputAttachments[i]` member of `VkSubpassDescription`. If the subpass

input variable is declared as an array of size N, it consumes N consecutive input attachments, starting with the index specified. There **must** not be more than one input variable with the same `InputAttachmentIndex` whether explicitly declared or implied by an array declaration. The number of available input attachment indices is given by the `maxPerStageDescriptorInputAttachments` member of the `VkPhysicalDeviceLimits` structure.

Variables identified with the `InputAttachmentIndex` **must** only be used by a fragment stage. The basic data type (floating-point, integer, unsigned integer) of the subpass input **must** match the basic format of the corresponding input attachment, or the values of subpass loads from these variables are undefined.

See [Input Attachment](#) for more details.

## 14.5. Shader Resource Interface

When a shader stage accesses buffer or image resources, as described in the [Resource Descriptors](#) section, the shader resource variables **must** be matched with the [pipeline layout](#) that is provided at pipeline creation time.

The set of shader resources that form the *shader resource interface* for a stage are the variables statically used by `OpEntryPoint` with the storage class of `Uniform`, `UniformConstant`, or `PushConstant`. For the fragment shader, this includes the [fragment input attachment interface](#).

The shader resource interface consists of two sub-interfaces: the push constant interface and the descriptor set interface.

### 14.5.1. Push Constant Interface

The shader variables defined with a storage class of `PushConstant` that are statically used by the shader entry points for the pipeline define the *push constant interface*. They **must** be:

- typed as `OpTypeStruct`,
- identified with a `Block` decoration, and
- laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in [Offset and Stride Assignment](#).

There **must** be no more than one push constant block statically used per shader entry point.

Each statically used member of a push constant block **must** be placed at an `Offset` such that the entire member is entirely contained within the `VkPushConstantRange` for each `OpEntryPoint` that uses it, and the `stageFlags` for that range **must** specify the appropriate `VkShaderStageFlagBits` for that stage. The `Offset` decoration for any member of a push constant block **must** not cause the space required for that member to extend outside the range [0, `maxPushConstantsSize`).

Any member of a push constant block that is declared as an array **must** only be accessed with *dynamically uniform* indices.

## 14.5.2. Descriptor Set Interface

The *descriptor set interface* is comprised of the shader variables with the storage class of `StorageBuffer`, `Uniform` or `UniformConstant` (including the variables in the [fragment input attachment interface](#)) that are statically used by the shader entry points for the pipeline.

These variables **must** have `DescriptorSet` and `Binding` decorations specified, which are assigned and matched with the `VkDescriptorSetLayout` objects in the pipeline layout as described in [DescriptorSet and Binding Assignment](#).

The `Image Format` of an `OpTypeImage` declaration **must** not be `Unknown`, for variables which are used for `OpImageRead`, `OpImageSparseRead`, or `OpImageWrite` operations, except under the following conditions:

- For `OpImageWrite`, if the `shaderStorageImageWriteWithoutFormat` feature is enabled and the shader module declares the `StorageImageWriteWithoutFormat` capability.
- For `OpImageRead` or `OpImageSparseRead`, if the `shaderStorageImageReadWithoutFormat` feature is enabled and the shader module declares the `StorageImageReadWithoutFormat` capability.
- For `OpImageRead`, if `Dim` is `SubpassData` (indicating a read from an input attachment).

The `Image Format` of an `OpTypeImage` declaration **must** not be `Unknown`, for variables which are used for `OpAtomic*` operations.

Variables identified with the `Uniform` storage class are used to access transparent buffer backed resources. Such variables **must** be:

- typed as `OpTypeStruct`, or an array of this type,
- identified with a `Block` or `BufferBlock` decoration, and
- laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in [Offset and Stride Assignment](#).

Variables identified with the `StorageBuffer` storage class are used to access transparent buffer backed resources. Such variables **must** be:

- typed as `OpTypeStruct`, or an array of this type,
- identified with a `Block` decoration, and
- laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in [Offset and Stride Assignment](#).

The `Offset` decoration for any member of a `Block`-decorated variable in the `Uniform` storage class **must** not cause the space required for that variable to extend outside the range  $[0, \text{maxUniformBufferRange}]$ . The `Offset` decoration for any member of a `Block`-decorated variable in the `StorageBuffer` storage class **must** not cause the space required for that variable to extend outside the range  $[0, \text{maxStorageBufferRange}]$ .

Variables identified with the `Uniform` storage class **can** also be used to access transparent descriptor set backed resources when the variable is assigned to a descriptor set layout binding with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`. In this case the variable **must** be

typed as `OpTypeStruct` and **cannot** be aggregated into arrays of that type. Further, the `Offset` decoration for any member of such a variable **must** not cause the space required for that variable to extend outside the range [0,`maxInlineUniformBlockSize`).

Variables identified with a storage class of `UniformConstant` and a decoration of `InputAttachmentIndex` **must** be declared as described in [Fragment Input Attachment Interface](#).

SPIR-V variables decorated with a descriptor set and binding that identify a [combined image sampler descriptor](#) **can** have a type of `OpTypeImage`, `OpTypeSampler` (`Sampled=1`), or `OpTypeSampledImage`.

Arrays of any of these types **can** be indexed with *constant integral expressions*. The following features **must** be enabled and capabilities **must** be declared in order to index such arrays with dynamically uniform or non-uniform indices:

- Storage images (except storage texel buffers and input attachments):
  - Dynamically uniform: `shaderStorageImageArrayDynamicIndexing` and `StorageImageArrayDynamicIndexing`
  - Non-uniform: `shaderStorageImageArrayNonUniformIndexing` and `StorageImageArrayNonUniformIndexingEXT`
- Storage texel buffers:
  - Dynamically uniform: `shaderStorageTexelBufferArrayDynamicIndexing` and `StorageTexelBufferArrayDynamicIndexingEXT`
  - Non-uniform: `shaderStorageTexelBufferArrayNonUniformIndexing` and `StorageTexelBufferArrayNonUniformIndexingEXT`
- Input attachments:
  - Dynamically uniform: `shaderInputAttachmentArrayDynamicIndexing` and `InputAttachmentArrayDynamicIndexingEXT`
  - Non-uniform: `shaderInputAttachmentArrayNonUniformIndexing` and `InputAttachmentArrayNonUniformIndexingEXT`
- Sampled images (except uniform texel buffers):
  - Dynamically uniform: `shaderSampledImageArrayDynamicIndexing` and `SampledImageArrayDynamicIndexing`
  - Non-uniform: `shaderSampledImageArrayNonUniformIndexing` and `SampledImageArrayNonUniformIndexingEXT`
- Uniform texel buffers:
  - Dynamically uniform: `shaderUniformTexelBufferArrayDynamicIndexing` and `UniformTexelBufferArrayDynamicIndexingEXT`
  - Non-uniform: `shaderUniformTexelBufferArrayNonUniformIndexing` and `UniformTexelBufferArrayNonUniformIndexingEXT`
- Uniform buffers:
  - Dynamically uniform: `shaderUniformBufferArrayDynamicIndexing` and `UniformBufferArrayDynamicIndexing`
  - Non-uniform: `shaderUniformBufferArrayNonUniformIndexing` and `UniformBufferArrayNonUniformIndexingEXT`
- Storage buffers:

- Dynamically uniform: `shaderStorageBufferArrayDynamicIndexing` and `StorageBufferArrayDynamicIndexing`
- Non-uniform: `shaderStorageBufferArrayNonUniformIndexing` and `StorageBufferArrayNonUniformIndexingEXT`
- Acceleration structures:
  - No additional capabilities needed.

If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not dynamically uniform, then the corresponding non-uniform indexing feature **must** be enabled and the capability **must** be declared. If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not uniform, then the corresponding dynamic indexing or non-uniform feature **must** be enabled and the capability **must** be declared.

If the combined image sampler enables sampler Y'C<sub>B</sub>C<sub>R</sub> conversion or samples a [subsampled image](#), it **must** be indexed only by constant integral expressions when aggregated into arrays in shader code, irrespective of the `shaderSampledImageArrayDynamicIndexing` feature.

*Table 19. Shader Resource and Descriptor Type Correspondence*

Resource type	Descriptor Type
sampler	<code>VK_DESCRIPTOR_TYPE_SAMPLER</code> or <code>VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</code>
sampled image	<code>VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE</code> or <code>VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</code>
storage image	<code>VK_DESCRIPTOR_TYPE_STORAGE_IMAGE</code>
combined image sampler	<code>VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</code>
uniform texel buffer	<code>VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER</code>
storage texel buffer	<code>VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER</code>
uniform buffer	<code>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER</code> or <code>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC</code>
storage buffer	<code>VK_DESCRIPTOR_TYPE_STORAGE_BUFFER</code> or <code>VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC</code>
input attachment	<code>VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT</code>
inline uniform block	<code>VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT</code>
acceleration structure	<code>VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV</code>

*Table 20. Shader Resource and Storage Class Correspondence*

Resource type	Storage Class	Type	Decoration(s) <sup>1</sup>
sampler	<code>UniformConstant</code>	<code>OpTypeSampler</code>	
sampled image	<code>UniformConstant</code>	<code>OpTypeImage (Sampled=1)</code>	
storage image	<code>UniformConstant</code>	<code>OpTypeImage (Sampled=2)</code>	
combined image sampler	<code>UniformConstant</code>	<code>OpTypeSampledImage</code> <code>OpTypeImage (Sampled=1)</code> <code>OpTypeSampler</code>	

Resource type	Storage Class	Type	Decoration(s) <sup>1</sup>
uniform texel buffer	UniformConstant	OpTypeImage (Dim=Buffer, Sampled=1)	
storage texel buffer	UniformConstant	OpTypeImage (Dim=Buffer, Sampled=2)	
uniform buffer	Uniform	OpTypeStruct	Block, Offset, (ArrayStride), (MatrixStride)
storage buffer	Uniform	OpTypeStruct	BufferBlock, Offset, (ArrayStride), (MatrixStride)
	StorageBuffer		Block, Offset, (ArrayStride), (MatrixStride)
input attachment	UniformConstant	OpTypeImage (Dim =SubpassData, Sampled=2)	InputAttachmentIndex
inline uniform block	Uniform	OpTypeStruct	Block, Offset, (ArrayStride), (MatrixStride)

<sup>1</sup>

in addition to [DescriptorSet](#) and [Binding](#)

### 14.5.3. DescriptorSet and Binding Assignment

A variable decorated with a [DescriptorSet](#) decoration of s and a [Binding](#) decoration of b indicates that this variable is associated with the [VkDescriptorSetLayoutBinding](#) that has a [binding](#) equal to b in [pSetLayouts\[s\]](#) that was specified in [VkPipelineLayoutCreateInfo](#).

[DescriptorSet](#) decoration values **must** be between zero and [maxBoundDescriptorSets](#) minus one, inclusive. [Binding](#) decoration values **can** be any 32-bit unsigned integer value, as described in [Descriptor Set Layout](#). Each descriptor set has its own binding name space.

If the [Binding](#) decoration is used with an array, the entire array is assigned that binding value. The array **must** be a single-dimensional array and size of the array **must** be no larger than the number of descriptors in the binding. If the array is runtime-sized, then array elements greater than or equal to the size of that binding in the bound descriptor set **must** not be used. If the array is runtime-sized, the [runtimeDescriptorArray](#) feature **must** be enabled and the [RuntimeDescriptorArrayEXT](#) capability **must** be declared. The index of each element of the array is referred to as the *arrayElement*. For the purposes of interface matching and descriptor set [operations](#), if a resource variable is not an array, it is treated as if it has an *arrayElement* of zero.

There is a limit on the number of resources of each type that **can** be accessed by a pipeline stage as shown in [Shader Resource Limits](#). The “Resources Per Stage” column gives the limit on the number each type of resource that **can** be statically used for an entry point in any given stage in a pipeline. The “Resource Types” column lists which resource types are counted against the limit. Some resource types count against multiple limits.

The pipeline layout **may** include descriptor sets and bindings which are not referenced by any variables statically used by the entry points for the shader stages in the binding’s [stageFlags](#).

However, if a variable assigned to a given [DescriptorSet](#) and [Binding](#) is statically used by the entry point for a shader stage, the pipeline layout **must** contain a descriptor set layout binding in that descriptor set layout and for that binding number, and that binding's [stageFlags](#) **must** include the appropriate [VkShaderStageFlagBits](#) for that stage. The variable **must** be of a valid resource type determined by its SPIR-V type and storage class, as defined in [Shader Resource and Storage Class Correspondence](#). The descriptor set layout binding **must** be of a corresponding descriptor type, as defined in [Shader Resource and Descriptor Type Correspondence](#).

*Note*

There are no limits on the number of shader variables that can have overlapping set and binding values in a shader; but which resources are [statically used](#) has an impact. If any shader variable identifying a resource is [statically used](#) in a shader, then the underlying descriptor bound at the declared set and binding must [support the declared type in the shader](#) when the shader executes.

If multiple shader variables are declared with the same set and binding values, and with the same underlying descriptor type, they can all be statically used within the same shader. However, accesses are not automatically synchronized, and [Aliased](#) decorations should be used to avoid data hazards (see [section 2.18.2 Aliasing in the SPIR-V specification](#)).



If multiple shader variables with the same set and binding values are declared in a single shader, but with different declared types, where any of those are not supported by the relevant bound descriptor, that shader can only be executed if the variables with the unsupported type are not statically used.

A noteworthy example of using multiple statically-used shader variables sharing the same descriptor set and binding values is a descriptor of type [VK\\_DESCRIPTOR\\_TYPE\\_COMBINED\\_IMAGE\\_SAMPLER](#) that has multiple corresponding shader variables in the [UniformConstant](#) storage class, where some could be [OpTypeImage](#), some could be [OpTypeSampler](#) ([Sampled=1](#)), and some could be [OpTypeSampledImage](#).

*Table 21. Shader Resource Limits*

Resources per Stage	Resource Types
<code>maxPerStageDescriptorSamplers</code> or <code>maxPerStageDescriptorUpdateAfterBindSamplers</code>	sampler
<code>maxPerStageDescriptorSampledImages</code> or <code>maxPerStageDescriptorUpdateAfterBindSampledImages</code>	combined image sampler
<code>maxPerStageDescriptorStorageImages</code> or <code>maxPerStageDescriptorUpdateAfterBindStorageImages</code>	sampled image
<code>maxPerStageDescriptorUniformBuffers</code> or <code>maxPerStageDescriptorUpdateAfterBindUniformBuffers</code>	combined image sampler
	uniform texel buffer
	storage image
	storage texel buffer
	uniform buffer
	uniform buffer dynamic

Resources per Stage	Resource Types
<code>maxPerStageDescriptorStorageBuffers</code> or <code>maxPerStageDescriptorUpdateAfterBindStorageBuffers</code>	storage buffer
<code>maxPerStageDescriptorInputAttachments</code> or <code>maxPerStageDescriptorUpdateAfterBindInputAttachments</code>	storage buffer dynamic input attachment <sup>1</sup>
<code>maxPerStageDescriptorInlineUniformBlocks</code> or <code>maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks</code>	inline uniform block

1

Input attachments **can** only be used in the fragment shader stage

#### 14.5.4. Offset and Stride Assignment

All variables with a storage class of `Uniform`, `StorageBuffer`, or `PushConstant` **must** be explicitly laid out using the `Offset`, `ArrayStride`, and `MatrixStride` decorations.



*Note*

The numeric order of `Offset` decorations does not need to follow member declaration order.

#### Alignment Requirements

There are different alignment requirements depending on the specific resources and on the features enabled on the device.

The *scalar alignment* of the type of an `OpTypeStruct` member is defined recursively as follows:

- A scalar of size N has a scalar alignment of N.
- A vector or matrix type has a scalar alignment equal to that of its component type.
- An array type has a scalar alignment equal to that of its element type.
- A structure has a scalar alignment equal to the largest scalar alignment of any of its members.

The *base alignment* of the type of an `OpTypeStruct` member is defined recursively as follows:

- A scalar has a base alignment equal to its scalar alignment.
- A two-component vector has a base alignment equal to twice its scalar alignment.
- A three- or four-component vector has a base alignment equal to four times its scalar alignment.
- An array has a base alignment equal to the base alignment of its element type.
- A structure has a base alignment equal to the largest base alignment of any of its members.
- A row-major matrix of C columns has a base alignment equal to the base alignment of a vector of C matrix components.
- A column-major matrix has a base alignment equal to the base alignment of the matrix column type.

The *extended alignment* of the type of an **OpTypeStruct** member is similarly defined as follows:

- A scalar, vector or matrix type has an extended alignment equal to its base alignment.
- An array or structure type has an extended alignment equal to the largest extended alignment of any of its members, rounded up to a multiple of 16.

A member is defined to *improperly straddle* if either of the following are true:

- It is a vector with total size less than or equal to 16 bytes, and has **Offset** decorations placing its first byte at F and its last byte at L, where  $\text{floor}(F / 16) \neq \text{floor}(L / 16)$ .
- It is a vector with total size greater than 16 bytes and has its **Offset** decorations placing its first byte at a non-integer multiple of 16.

## Standard Buffer Layout

Every member of an **OpTypeStruct** with storage class of **Uniform**, **StorageBuffer**, or **PushConstant** **must** be aligned according to the first matching rule as follows:

1. If the **scalarBlockLayout** feature is enabled on the device then every member **must** be aligned according to its scalar alignment.
2. All vectors **must** be aligned according to their scalar alignment.
3. If the **uniformBufferStandardLayout** feature is not enabled on the device, then any member of an **OpTypeStruct** with a storage class of **Uniform** and a decoration of **Block** **must** be aligned according to its extended alignment.
4. Every other member **must** be aligned according to its base alignment.

### Note



Even if scalar alignment is supported, it is generally more performant to use the *base alignment*.

The memory layout **must** obey the following rules:

- The **Offset** decoration of any member **must** be a multiple of its alignment.
- Any **ArrayStride** or **MatrixStride** decoration **must** be a multiple of the alignment of the array or matrix as defined above.

Unless the **scalarBlockLayout** feature is enabled on the device:

- Vectors **must** not improperly straddle, as defined above.
- The **Offset** decoration of a member **must** not place it between the end of a structure or an array and the next multiple of the alignment of that structure or array.

### Note



The **std430** layout in GLSL satisfies these rules for types using the base alignment. The **std140** layout satisfies the rules for types using the extended alignment.

## 14.6. Built-In Variables

Built-in variables are accessed in shaders by declaring a variable decorated with a `BuiltIn` SPIR-V decoration. The meaning of each `BuiltIn` decoration is as follows. In the remainder of this section, the name of a built-in is used interchangeably with a term equivalent to a variable decorated with that particular built-in. Built-ins that represent integer values **can** be declared as either signed or unsigned 32-bit integers.

### `BaryCoordNV`

The `BaryCoordNV` decoration **can** be used to decorate a fragment shader input variable. This variable will contain a three-component floating-point vector with barycentric weights that indicate the location of the fragment relative to the screen-space locations of vertices of its primitive, obtained using perspective interpolation.

The `BaryCoordNV` decoration **must** be used only within fragment shaders.

The variable decorated with `BaryCoordNV` **must** be declared using the `Input` storage class.

The variable decorated with `BaryCoordNV` **must** be declared as three-component vector of 32-bit floating-point values.

### `BaryCoordNoPerspAMD`

The `BaryCoordNoPerspAMD` decoration **can** be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using linear interpolation at the fragment's center. The K coordinate of the barycentric coordinates **can** be derived given the identity  $I + J + K = 1.0$ .

### `BaryCoordNoPerspNV`

The `BaryCoordNoPerspNV` decoration **can** be used to decorate a fragment shader input variable. This variable will contain a three-component floating-point vector with barycentric weights that indicate the location of the fragment relative to the screen-space locations of vertices of its primitive, obtained using linear interpolation.

The `BaryCoordNoPerspNV` decoration **must** be used only within fragment shaders.

The variable decorated with `BaryCoordNoPerspNV` **must** be declared using the `Input` storage class.

The variable decorated with `BaryCoordNoPerspNV` **must** be declared as three-component vector of 32-bit floating-point values.

### `BaryCoordNoPerspCentroidAMD`

The `BaryCoordNoPerspCentroidAMD` decoration **can** be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using linear interpolation at the centroid. The K coordinate of the barycentric coordinates **can** be derived given the identity  $I + J + K = 1.0$ .

### `BaryCoordNoPerspSampleAMD`

The `BaryCoordNoPerspCentroidAMD` decoration **can** be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using linear interpolation at each covered sample. The K coordinate of

the barycentric coordinates **can** be derived given the identity  $I + J + K = 1.0$ .

#### BaryCoordPullModelAMD

The **BaryCoordPullModelAMD** decoration **can** be used to decorate a fragment shader input variable. This variable will contain  $(1/W, 1/I, 1/J)$  evaluated at the fragment center and **can** be used to calculate gradients and then interpolate I, J, and W at any desired sample location.

#### BaryCoordSmoothAMD

The **BaryCoordSmoothAMD** decoration **can** be used to decorate a fragment shader input variable. This variable will contain the  $(I,J)$  pair of the barycentric coordinates corresponding to the fragment evaluated using perspective interpolation at the fragment's center. The K coordinate of the barycentric coordinates **can** be derived given the identity  $I + J + K = 1.0$ .

#### BaryCoordSmoothCentroidAMD

The **BaryCoordSmoothCentroidAMD** decoration **can** be used to decorate a fragment shader input variable. This variable will contain the  $(I,J)$  pair of the barycentric coordinates corresponding to the fragment evaluated using perspective interpolation at the centroid. The K coordinate of the barycentric coordinates **can** be derived given the identity  $I + J + K = 1.0$ .

#### BaryCoordSmoothSampleAMD

The **BaryCoordSmoothCentroidAMD** decoration **can** be used to decorate a fragment shader input variable. This variable will contain the  $(I,J)$  pair of the barycentric coordinates corresponding to the fragment evaluated using perspective interpolation at each covered sample. The K coordinate of the barycentric coordinates **can** be derived given the identity  $I + J + K = 1.0$ .

#### BaseInstance

Decorating a variable with the **BaseInstance** built-in will make that variable contain the integer value corresponding to the first instance that was passed to the command that invoked the current vertex shader invocation. **BaseInstance** is the **firstInstance** parameter to a *direct drawing command* or the **firstInstance** member of a structure consumed by an *indirect drawing command*.

The **BaseInstance** decoration **must** be used only within vertex shaders.

The variable decorated with **BaseInstance** **must** be declared using the input storage class.

The variable decorated with **BaseInstance** **must** be declared as a scalar 32-bit integer.

#### BaseVertex

Decorating a variable with the **BaseVertex** built-in will make that variable contain the integer value corresponding to the first vertex or vertex offset that was passed to the command that invoked the current vertex shader invocation. For *non-indexed drawing commands*, this variable is the **firstVertex** parameter to a *direct drawing command* or the **firstVertex** member of the structure consumed by an *indirect drawing command*. For *indexed drawing commands*, this variable is the **vertexOffset** parameter to a *direct drawing command* or the **vertexOffset** member of the structure consumed by an *indirect drawing command*.

The **BaseVertex** decoration **must** be used only within vertex shaders.

The variable decorated with **BaseVertex** **must** be declared using the input storage class.

The variable decorated with `BaseVertex` **must** be declared as a scalar 32-bit integer.

### `ClipDistance`

Decorating a variable with the `ClipDistance` built-in decoration will make that variable contain the mechanism for controlling user clipping. `ClipDistance` is an array such that the  $i^{\text{th}}$  element of the array specifies the clip distance for plane  $i$ . A clip distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the clip half-space, and a negative distance means the point is outside the clip half-space.

The `ClipDistance` decoration **must** be used only within mesh, vertex, fragment, tessellation control, tessellation evaluation, and geometry shaders.

In mesh or vertex shaders, any variable decorated with `ClipDistance` **must** be declared using the `Output` storage class.

In fragment shaders, any variable decorated with `ClipDistance` **must** be declared using the `Input` storage class.

In tessellation control, tessellation evaluation, or geometry shaders, any variable decorated with `ClipDistance` **must** not be in a storage class other than `Input` or `Output`.

Any variable decorated with `ClipDistance` **must** be declared as an array of 32-bit floating-point values.



#### *Note*

The array variable decorated with `ClipDistance` is explicitly sized by the shader.



#### *Note*

In the last vertex processing stage, these values will be linearly interpolated across the primitive and the portion of the primitive with interpolated distances less than 0 will be considered outside the clip volume. If `ClipDistance` is then used by a fragment shader, `ClipDistance` contains these linearly interpolated values.

### `ClipDistancePerViewNV`

Decorating a variable with the `ClipDistancePerViewNV` built-in decoration will make that variable contain the per-view clip distances. The per-view clip distances have the same semantics as `ClipDistance`.

The `ClipDistancePerViewNV` **must** be used only within mesh shaders.

Any variable decorated with `ClipDistancePerViewNV` **must** be declared using the `Output` storage class, and **must** also be decorated with the `PerViewNV` decoration.

Any variable decorated with `ClipDistancePerViewNV` **must** be declared as a two-dimensional array of 32-bit floating-point values.

### `CullDistance`

Decorating a variable with the `CullDistance` built-in decoration will make that variable contain the mechanism for controlling user culling. If any member of this array is assigned a negative

value for all vertices belonging to a primitive, then the primitive is discarded before rasterization.

The `CullDistance` decoration **must** be used only within mesh, vertex, fragment, tessellation control, tessellation evaluation, and geometry shaders.

In mesh or vertex shaders, any variable decorated with `CullDistance` **must** be declared using the `Output` storage class.

In fragment shaders, any variable decorated with `CullDistance` **must** be declared using the `Input` storage class.

In tessellation control, tessellation evaluation, or geometry shaders, any variable decorated with `CullDistance` **must** not be declared in a storage class other than input or output.

Any variable decorated with `CullDistance` **must** be declared as an array of 32-bit floating-point values.

*Note*



In fragment shaders, the values of the `CullDistance` array are linearly interpolated across each primitive.

*Note*



If `CullDistance` decorates an input variable, that variable will contain the corresponding value from the `CullDistance` decorated output variable from the previous shader stage.

## `CullDistancePerViewNV`

Decorating a variable with the `CullDistancePerViewNV` built-in decoration will make that variable contain the per-view cull distances. The per-view clip distances have the same semantics as `CullDistance`.

The `CullDistancePerViewNV` **must** be used only within mesh shaders.

Any variable decorated with `CullDistancePerViewNV` **must** be declared using the `Output` storage class, and **must** also be decorated with the `PerViewNV` decoration.

Any variable decorated with `CullDistancePerViewNV` **must** be declared as a two-dimensional array of 32-bit floating-point values.

## `DeviceIndex`

The `DeviceIndex` decoration **can** be applied to a shader input which will be filled with the device index of the physical device that is executing the current shader invocation. This value will be in the range  $[0, \max(1, \text{physicalDeviceCount})]$ , where `physicalDeviceCount` is the `physicalDeviceCount` member of `VkDeviceGroupCreateInfo`.

The `DeviceIndex` decoration **can** be used in any shader.

The variable decorated with `DeviceIndex` **must** be declared using the `Input` storage class.

The variable decorated with **DeviceIndex** **must** be declared as a scalar 32-bit integer.

### DrawIndex

Decorating a variable with the **DrawIndex** built-in will make that variable contain the integer value corresponding to the zero-based index of the drawing command that invoked the current task, mesh, or vertex shader invocation. For *indirect drawing commands*, **DrawIndex** begins at zero and increments by one for each draw command executed. The number of draw commands is given by the **drawCount** parameter. For *direct drawing commands*, **DrawIndex** is always zero. **DrawIndex** is dynamically uniform.

The **DrawIndex** decoration **must** be used only within task, mesh or vertex shaders.

The variable decorated with **DrawIndex** **must** be declared using the **Input** storage class.

The variable decorated with **DrawIndex** **must** be declared as a scalar 32-bit integer.

When task or mesh shaders are used, only the first active stage will have proper access to the variable, other stages will have undefined values.

### FragCoord

Decorating a variable with the **FragCoord** built-in decoration will make that variable contain the framebuffer coordinate  $(x, y, z, \frac{1}{w})$  of the fragment being processed. The  $(x, y)$  coordinate  $(0, 0)$  is the upper left corner of the upper left pixel in the framebuffer.

When **Sample Shading** is enabled, the  $x$  and  $y$  components of **FragCoord** reflect the location of one of the samples corresponding to the shader invocation.

Otherwise, the  $x$  and  $y$  components of **FragCoord** reflect the location of the center of the fragment.

The  $z$  component of **FragCoord** is the interpolated depth value of the primitive.

The  $w$  component is the interpolated  $\frac{1}{w}$ .

The **FragCoord** decoration **must** be used only within fragment shaders.

The variable decorated with **FragCoord** **must** be declared using the **Input** storage class.

The **Centroid** interpolation decoration is ignored, but allowed, on **FragCoord**.

The variable decorated with **FragCoord** **must** be declared as a four-component vector of 32-bit floating-point values.

### FragDepth

To have a shader supply a fragment-depth value, the shader **must** declare the **DepthReplacing** execution mode. Such a shader's fragment-depth value will come from the variable decorated with the **FragDepth** built-in decoration.

This value will be used for any subsequent depth testing performed by the implementation or writes to the depth attachment.

The **FragDepth** decoration **must** be used only within fragment shaders.

The variable decorated with `FragDepth` **must** be declared using the `Output` storage class.

The variable decorated with `FragDepth` **must** be declared as a scalar 32-bit floating-point value.

### `FragInvocationCountEXT`

Decorating a variable with the `FragInvocationCountEXT` built-in decoration will make that variable contain the maximum number of fragment shader invocations for the fragment, as determined by `minSampleShading`.

The `FragInvocationCountEXT` decoration **must** be used only within fragment shaders and the `FragmentDensityEXT` capability **must** be declared.

If `Sample Shading` is not enabled, `FragInvocationCountEXT` will be filled with a value of 1.

The variable decorated with `FragInvocationCountEXT` **must** be declared using the `Input` storage class.

The variable decorated with `FragInvocationCountEXT` **must** be declared as a scalar 32-bit integer.

### `FragSizeEXT`

Decorating a variable with the `FragSizeEXT` built-in decoration will make that variable contain the dimensions in pixels of the `area` that the fragment covers for that invocation.

The `FragSizeEXT` decoration **must** be used only within fragment shaders and the `FragmentDensityEXT` capability **must** be declared.

If fragment density map is not enabled, `FragSizeEXT` will be filled with a value of (1,1).

The variable decorated with `FragSizeEXT` **must** be declared using the `Input` storage class.

The variable decorated with `FragSizeEXT` **must** be declared as a two-component vector of 32-bit integers.

### `FragStencilRefEXT`

Decorating a variable with the `FragStencilRefEXT` built-in decoration will make that variable contain the new stencil reference value for all samples covered by the fragment. This value will be used as the stencil reference value used in stencil testing.

To write to `FragStencilRefEXT`, a shader **must** declare the `StencilRefReplacingEXT` execution mode. If a shader declares the `StencilRefReplacingEXT` execution mode and there is an execution path through the shader that does not set `FragStencilRefEXT`, then the fragment's stencil reference value is undefined for executions of the shader that take that path.

The `FragStencilRefEXT` decoration **must** be used only within fragment shaders.

The variable decorated with `FragStencilRefEXT` **must** be declared using the `Output` storage class.

The variable decorated with `FragStencilRefEXT` **must** be declared as a scalar integer value. Only the least significant `s` bits of the integer value of the variable decorated with `FragStencilRefEXT` are considered for stencil testing, where `s` is the number of bits in the stencil framebuffer attachment, and higher order bits are discarded.

## FragmentSizeNV

Decorating a variable with the `FragmentSizeNV` built-in decoration will make that variable contain the width and height of the fragment.

The `FragmentSizeNV` decoration **must** be used only within fragment shaders.

The variable decorated with `FragmentSizeNV` **must** be declared using the `Input` storage class.

The variable decorated with `FragmentSizeNV` **must** be declared as a two-component vector of 32-bit integers.

## FrontFacing

Decorating a variable with the `FrontFacing` built-in decoration will make that variable contain whether the fragment is front or back facing. This variable is non-zero if the current fragment is considered to be part of a `front-facing` polygon primitive or of a non-polygon primitive and is zero if the fragment is considered to be part of a back-facing polygon primitive.

The `FrontFacing` decoration **must** be used only within fragment shaders.

The variable decorated with `FrontFacing` **must** be declared using the `Input` storage class.

The variable decorated with `FrontFacing` **must** be declared as a boolean.

## FullyCoveredEXT

Decorating a variable with the `FullyCoveredEXT` built-in decoration will make that variable indicate whether the `fragment area` is fully covered by the generating primitive. This variable is non-zero if conservative rasterization is enabled and the current fragment area is fully covered by the generating primitive, and is zero if the fragment is not covered or partially covered, or conservative rasterization is disabled.

The `FullyCoveredEXT` decoration **must** be used only within fragment shaders and the `FragmentFullyCoveredEXT` capability **must** be declared.

The variable decorated with `FullyCoveredEXT` **must** be declared using the `Input` storage class.

The variable decorated with `FullyCoveredEXT` **must** be declared as a boolean.

If the implementation supports `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage` and the `PostDepthCoverage` execution mode is specified the `SampleMask` built-in input variable will reflect the coverage after the early per-fragment depth and stencil tests are applied. If `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage` is not supported the `PostDepthCoverage` execution mode **must** not be specified.

## GlobalInvocationId

Decorating a variable with the `GlobalInvocationId` built-in decoration will make that variable contain the location of the current invocation within the global workgroup. Each component is equal to the index of the local workgroup multiplied by the size of the local workgroup plus `LocalInvocationId`.

The `GlobalInvocationId` decoration **must** be used only within task, mesh, or compute shaders.

The variable decorated with `GlobalInvocationId` **must** be declared using the `Input` storage class.

The variable decorated with `GlobalInvocationId` **must** be declared as a three-component vector of 32-bit integers.

### HelperInvocation

Decorating a variable with the `HelperInvocation` built-in decoration will make that variable contain whether the current invocation is a helper invocation. This variable is non-zero if the current fragment being shaded is a helper invocation and zero otherwise. A helper invocation is an invocation of the shader that is produced to satisfy internal requirements such as the generation of derivatives.

The `HelperInvocation` decoration **must** be used only within fragment shaders.

The variable decorated with `HelperInvocation` **must** be declared using the `Input` storage class.

The variable decorated with `HelperInvocation` **must** be declared as a boolean.

#### *Note*



It is very likely that a helper invocation will have a value of `SampleMask` fragment shader input value that is zero.

### HitKindNV

A variable decorated with the `HitKindNV` decoration will describe the intersection that triggered the execution of the current shader. The values are determined by the intersection shader.

The `HitKindNV` decoration **must** only be used in any-hit and closest hit shaders.

Any variable decorated with `HitKindNV` **must** be declared using the `Input` storage class.

Any variable decorated with `HitKindNV` **must** be declared as a scalar 32-bit integer.

### HitTnv

A variable decorated with the `HitTnv` decoration is equivalent to a variable decorated with the `RayTmaxNV` decoration.

The `HitTnv` decoration **must** only be used in any-hit and closest hit shaders.

Any variable decorated with `HitTnv` **must** be declared using the `Input` storage class.

Any variable decorated with `HitTnv` **must** be declared as a scalar 32-bit floating-point value.

### IncomingRayFlagsNV

A variable with the `IncomingRayFlagsNV` decoration will contain the ray flags passed in to the trace call that invoked this particular shader.

The `IncomingRayFlagsNV` decoration **must** only be used in the intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `IncomingRayFlagsNV` **must** be declared using the `Input` storage class.

Any variable decorated with `IncomingRayFlagsNV` **must** be declared as a scalar 32-bit integer.

#### `InstanceCustomIndexNV`

A variable decorated with the `InstanceCustomIndexNV` decoration will contain the application-defined value of the instance that intersects the current ray. Only the lower 24 bits are valid, the upper 8 bits will be ignored.

The `InstanceCustomIndexNV` decoration **must** only be used in the intersection, any-hit, and closest hit shaders.

Any variable decorated with `InstanceCustomIndexNV` **must** be declared using the `Input` storage class.

Any variable decorated with `InstanceCustomIndexNV` **must** be declared as a scalar 32-bit integer.

#### `InstanceId`

Decorating a variable in an intersection, any-hit, or closest hit shader with the `InstanceId` decoration will make that variable contain the index of the instance that intersects the current ray.

The `InstanceId` decoration **must** be used only within intersection, any-hit, or closest hit shaders.

The variable decorated with `InstanceId` **must** be declared using the `Input` storage class.

The variable decorated with `InstanceId` **must** be declared as a scalar 32-bit integer.

#### `InvocationId`

Decorating a variable with the `InvocationId` built-in decoration will make that variable contain the index of the current shader invocation in a geometry shader, or the index of the output patch vertex in a tessellation control shader.

In a geometry shader, the index of the current shader invocation ranges from zero to the number of `instances` declared in the shader minus one. If the instance count of the geometry shader is one or is not specified, then `InvocationId` will be zero.

The `InvocationId` decoration **must** be used only within tessellation control and geometry shaders.

The variable decorated with `InvocationId` **must** be declared using the `Input` storage class.

The variable decorated with `InvocationId` **must** be declared as a scalar 32-bit integer.

#### `InvocationsPerPixelNV`

Decorating a variable with the `InvocationsPerPixelNV` built-in decoration will make that variable contain the maximum number of fragment shader invocations per pixel, as derived from the effective shading rate for the fragment. If a primitive does not fully cover a pixel, the number of fragment shader invocations for that pixel **may** be less than the value of `InvocationsPerPixelNV`. If the shading rate indicates a fragment covering multiple pixels, then `InvocationsPerPixelNV` will be one.

The `InvocationsPerPixelNV` decoration **must** be used only within fragment shaders.

The variable decorated with `InvocationsPerPixelNV` **must** be declared using the `Input` storage class.

The variable decorated with `InvocationsPerPixelNV` **must** be declared as a scalar 32-bit integer.

### InstanceIndex

Decorating a variable in a vertex shader with the `InstanceIndex` built-in decoration will make that variable contain the index of the instance that is being processed by the current vertex shader invocation. `InstanceIndex` begins at the `firstInstance` parameter to `vkCmdDraw` or `vkCmdDrawIndexed` or at the `firstInstance` member of a structure consumed by `vkCmdDrawIndirect` or `vkCmdDrawIndexedIndirect`.

The `InstanceIndex` decoration **must** be used only within vertex shaders.

The variable decorated with `InstanceIndex` **must** be declared using the `Input` storage class.

The variable decorated with `InstanceIndex` **must** be declared as a scalar 32-bit integer.

### LaunchIDNV

A variable decorated with the `LaunchIDNV` decoration will specify the index of the work item being process. One work item is generated for each of the `width × height × depth` items dispatched by a `vkCmdTraceRaysNV` command. All shader invocations inherit the same value for variables decorated with `LaunchIDNV`.

The `LaunchIDNV` decoration **must** only be used within the ray generation, intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `LaunchIDNV` **must** be declared using the `Input` storage class.

Any variable decorated with `LaunchIDNV` **must** be declared as a three-component vector of 32-bit integer values.

### LaunchSizeNV

A variable decorated with the `LaunchSizeNV` decoration will contain the `width`, `height`, and `depth` dimensions passed to the `vkCmdTraceRaysNV` command that initiated this shader execution. The `width` is in the first component, the `height` is in the second component, and the `depth` is in the third component.

The `LaunchSizeNV` decoration **must** only be used within ray generation, intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `LaunchSizeNV` **must** be declared using the `Input` storage class.

Any variable decorated with `LaunchSizeNV` **must** be declared as a three-component vector of 32-bit integer values.

### Layer

Decorating a variable with the `Layer` built-in decoration will make that variable contain the select layer of a multi-layer framebuffer attachment.

In a mesh, vertex, tessellation evaluation, or geometry shader, any variable decorated with `Layer`

can be written with the framebuffer layer index to which the primitive produced by that shader will be directed.

The last active *vertex processing stage* (in pipeline order) controls the `Layer` that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the `Layer`.

If the last active vertex processing stage shader entry point's interface does not include a variable decorated with `Layer`, then the first layer is used. If a vertex processing stage shader entry point's interface includes a variable decorated with `Layer`, it **must** write the same value to `Layer` for all output vertices of a given primitive. If the `Layer` value is less than 0 or greater than or equal to the number of layers in the framebuffer, then primitives **may** still be rasterized, fragment shaders **may** be executed, and the framebuffer values for all layers are undefined.

The `Layer` decoration **must** be used only within mesh, vertex, tessellation evaluation, geometry, and fragment shaders.

In a mesh, vertex, tessellation evaluation, or geometry shader, any variable decorated with `Layer` **must** be declared using the `Output` storage class. If such a variable is also decorated with `ViewportRelativeNV`, then the `ViewportIndex` is added to the layer that is used for rendering and that is made available in the fragment shader. If the shader writes to a variable decorated `ViewportMaskNV`, then the layer selected has a different value for each viewport a primitive is rendered to.

In a fragment shader, a variable decorated with `Layer` contains the layer index of the primitive that the fragment invocation belongs to.

In a fragment shader, any variable decorated with `Layer` **must** be declared using the `Input` storage class.

Any variable decorated with `Layer` **must** be declared as a scalar 32-bit integer.

## `LayerPerViewNV`

Decorating a variable with the `LayerPerViewNV` built-in decoration will make that variable contain the per-view layer information. The per-view layer has the same semantics as `Layer`, for each view.

The `LayerPerViewNV` **must** only be used within mesh shaders.

Any variable decorated with `LayerPerViewNV` **must** be declared using the `Output` storage class, and **must** also be decorated with the `PerViewNV` decoration.

Any variable decorated with `LayerPerViewNV` **must** be declared as an array of scalar 32-bit integer values.

## `LocalInvocationId`

Decorating a variable with the `LocalInvocationId` built-in decoration will make that variable contain the location of the current task, mesh, or compute shader invocation within the local workgroup. Each component ranges from zero through to the size of the workgroup in that dimension minus one.

The `LocalInvocationId` decoration **must** be used only within task, mesh, or compute shaders.

The variable decorated with `LocalInvocationId` **must** be declared using the `Input` storage class.

The variable decorated with `LocalInvocationId` **must** be declared as a three-component vector of 32-bit integers.

*Note*

If the size of the workgroup in a particular dimension is one, then the `LocalInvocationId` in that dimension will be zero. If the workgroup is effectively two-dimensional, then `LocalInvocationId.z` will be zero. If the workgroup is effectively one-dimensional, then both `LocalInvocationId.y` and `LocalInvocationId.z` will be zero.



### `LocalInvocationIndex`

Decorating a variable with the `LocalInvocationIndex` built-in decoration will make that variable contain a one-dimensional representation of `LocalInvocationId`. This is computed as:

```
LocalInvocationIndex =  
    LocalInvocationId.z * WorkgroupSize.x * WorkgroupSize.y +  
    LocalInvocationId.y * WorkgroupSize.x +  
    LocalInvocationId.x;
```

The `LocalInvocationIndex` decoration **must** be used only within task, mesh, or compute shaders.

The variable decorated with `LocalInvocationIndex` **must** be declared using the `Input` storage class.

The variable decorated with `LocalInvocationIndex` **must** be declared as a scalar 32-bit integer.

### `MeshViewCountNV`

Decorating a variable with the `MeshViewCountNV` built-in decoration will make that variable contain the number of views processed by the current mesh or task shader invocations.

The `MeshViewCountNV` decoration **must** only be used in task and mesh shaders.

Any variable decorated with `MeshViewCountNV` **must** be declared using the `Input` storage class.

Any variable decorated with `MeshViewCountNV` **must** be declared as a scalar 32-bit integer.

### `MeshViewIndicesNV`

Decorating a variable with the `MeshViewIndicesNV` built-in decoration will make that variable contain the mesh view indices. The mesh view indices is an array of values where each element holds the view number of one of the views being processed by the current mesh or task shader invocations. The array elements with indices great than or equal to `MeshViewCountNV` are undefined. If the value of `MeshViewIndicesNV[i]` is `j`, then any outputs decorated with `PerViewNV` will take on the value of array element `i` when processing primitives for view index `j`.

The `MeshViewIndicesNV` decoration **must** only be used in task and mesh shaders.

Any variable decorated with `MeshViewIndicesNV` **must** be declared using the `Input` storage class.

Any variable decorated with `MeshViewIndicesNV` **must** be declared as an array of scalar 32-bit integers.

### `NumSubgroups`

Decorating a variable with the `NumSubgroups` built-in decoration will make that variable contain the number of subgroups in the local workgroup.

The `NumSubgroups` decoration **must** be used only within task, mesh, or compute shaders.

The variable decorated with `NumSubgroups` **must** be declared using the `Input` storage class.

The object decorated with `NumSubgroups` **must** be declared as a scalar 32-bit integer.

### `NumWorkgroups`

Decorating a variable with the `NumWorkgroups` built-in decoration will make that variable contain the number of local workgroups that are part of the dispatch that the invocation belongs to. Each component is equal to the values of the workgroup count parameters passed into the dispatch commands.

The `NumWorkgroups` decoration **must** be used only within compute shaders.

The variable decorated with `NumWorkgroups` **must** be declared using the `Input` storage class.

The variable decorated with `NumWorkgroups` **must** be declared as a three-component vector of 32-bit integers.

### `ObjectRayDirectionNV`

A variable decorated with the `ObjectRayDirectionNV` decoration will specify the direction of the ray being processed, in object space.

The `ObjectRayDirectionNV` decoration **must** only be used within intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `ObjectRayDirectionNV` **must** be declared using the `Input` storage class.

Any variable decorated with `ObjectRayDirectionNV` **must** be declared as a three-component vector of 32-bit floating-point values.

### `ObjectRayOriginNV`

A variable decorated with the `ObjectRayOriginNV` decoration will specify the origin of the ray being processed, in object space.

The `ObjectRayOriginNV` decoration **must** only be used within intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `ObjectRayOriginNV` **must** be declared using the `Input` storage class.

Any variable decorated with `ObjectRayOriginNV` **must** be declared as a three-component vector of 32-bit floating-point values.

## ObjectToWorldNV

A variable decorated with the `ObjectToWorldNV` decoration will contain the current object-to-world transformation matrix, which is determined by the instance of the current intersection.

The `ObjectToWorldNV` decoration **must** only be used within intersection, any-hit, and closest hit shaders.

Any variable decorated with `ObjectToWorldNV` **must** be declared using the `Input` storage class.

Any variable decorated with `ObjectToWorldNV` **must** be declared as a matrix with four columns of three-component vectors of 32-bit floating-point values.

## PatchVertices

Decorating a variable with the `PatchVertices` built-in decoration will make that variable contain the number of vertices in the input patch being processed by the shader. A single tessellation control or tessellation evaluation shader **can** read patches of differing sizes, so the value of the `PatchVertices` variable **may** differ between patches.

The `PatchVertices` decoration **must** be used only within tessellation control and tessellation evaluation shaders.

The variable decorated with `PatchVertices` **must** be declared using the `Input` storage class.

The variable decorated with `PatchVertices` **must** be declared as a scalar 32-bit integer.

## PointCoord

Decorating a variable with the `PointCoord` built-in decoration will make that variable contain the coordinate of the current fragment within the point being rasterized, normalized to the size of the point with origin in the upper left corner of the point, as described in [Basic Point Rasterization](#). If the primitive the fragment shader invocation belongs to is not a point, then the variable decorated with `PointCoord` contains an undefined value.

The `PointCoord` decoration **must** be used only within fragment shaders.

The variable decorated with `PointCoord` **must** be declared using the `Input` storage class.

The variable decorated with `PointCoord` **must** be declared as two-component vector of 32-bit floating-point values.

### Note



Depending on how the point is rasterized, `PointCoord` **may** never reach (0,0) or (1,1).

## PointSize

Decorating a variable with the `PointSize` built-in decoration will make that variable contain the size of point primitives. The value written to the variable decorated with `PointSize` by the last vertex processing stage in the pipeline is used as the framebuffer-space size of points produced by rasterization.

The `PointSize` decoration **must** be used only within mesh, vertex, tessellation control,

tessellation evaluation, and geometry shaders.

In a mesh or vertex shader, any variable decorated with **PointSize** **must** be declared using the **Output** storage class.

In a tessellation control, tessellation evaluation, or geometry shader, any variable decorated with **PointSize** **must** be declared using either the **Input** or **Output** storage class.

Any variable decorated with **PointSize** **must** be declared as a scalar 32-bit floating-point value.

*Note*



When **PointSize** decorates a variable in the **Input** storage class, it contains the data written to the output variable decorated with **PointSize** from the previous shader stage.

## Position

Decorating a variable with the **Position** built-in decoration will make that variable contain the position of the current vertex. In the last vertex processing stage, the value of the variable decorated with **Position** is used in subsequent primitive assembly, clipping, and rasterization operations.

The **Position** decoration **must** be used only within mesh, vertex, tessellation control, tessellation evaluation, and geometry shaders.

In a mesh or vertex shader, any variable decorated with **Position** **must** be declared using the **Output** storage class.

In a tessellation control, tessellation evaluation, or geometry shader, any variable decorated with **Position** **must** not be declared in a storage class other than **Input** or **Output**.

Any variable decorated with **Position** **must** be declared as a four-component vector of 32-bit floating-point values.

*Note*



When **Position** decorates a variable in the **Input** storage class, it contains the data written to the output variable decorated with **Position** from the previous shader stage.

## PositionPerViewNV

Decorating a variable with the **PositionPerViewNV** built-in decoration will make that variable contain the position of the current vertex, for each view.

The **PositionPerViewNV** decoration **must** be used only within mesh, vertex, tessellation control, tessellation evaluation, and geometry shaders.

In a vertex shader, any variable decorated with **PositionPerViewNV** **must** be declared using the **Output** storage class.

In a tessellation control, tessellation evaluation, or geometry shader, any variable decorated with **PositionPerViewNV** **must** not be declared in a storage class other than input or output.

Any variable decorated with `PositionPerViewNV` **must** be declared as an array of four-component vector of 32-bit floating-point values with at least as many elements as the maximum view in the subpass's view mask plus one. The array **must** be indexed by a constant or specialization constant.

Elements of the array correspond to views in a multiview subpass, and those elements corresponding to views in the view mask of the subpass the shader is compiled against will be used as the position value for those views. For the final vertex processing stage in the pipeline, values written to an output variable decorated with `PositionPerViewNV` are used in subsequent primitive assembly, clipping, and rasterization operations, as with `Position`. `PositionPerViewNV` output in an earlier vertex processing stage is available as an input in the subsequent vertex processing stage.

If a shader is compiled against a subpass that has the `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX` bit set, then the position values for each view **must** not differ in any component other than the X component. If the values do differ, one will be chosen in an implementation-dependent manner.

### `PrimitiveCountNV`

Decorating a variable with the `PrimitiveCountNV` decoration will make that variable contain the primitive count. The primitive count specifies the number of primitives in the output mesh produced by the mesh shader that will be processed by subsequent pipeline stages.

The `PrimitiveCountNV` decoration **must** only be used in mesh shaders.

Any variable decorated with `PrimitiveCountNV` **must** be declared using the `Output` storage class.

Any variable decorated with `PrimitiveCountNV` **must** be declared as a scalar 32-bit integer.

### `PrimitiveId`

Decorating a variable with the `PrimitiveId` built-in decoration will make that variable contain the index of the current primitive.

The index of the first primitive generated by a drawing command is zero, and the index is incremented after every individual point, line, or triangle primitive is processed.

For triangles drawn as points or line segments (see [Polygon Mode](#)), the primitive index is incremented only once, even if multiple points or lines are eventually drawn.

Variables decorated with `PrimitiveId` are reset to zero between each instance drawn.

Restarting a primitive topology using primitive restart has no effect on the value of variables decorated with `PrimitiveId`.

In tessellation control and tessellation evaluation shaders, it will contain the index of the patch within the current set of rendering primitives that correspond to the shader invocation.

In a geometry shader, it will contain the number of primitives presented as input to the shader since the current set of rendering primitives was started.

In a fragment shader, it will contain the primitive index written by the geometry shader if a

geometry shader is present, or with the value that would have been presented as input to the geometry shader had it been present.

In an intersection, any-hit, or closest hit shader, it will contain the index within the geometry of the triangle or bounding box being processed.

If a geometry shader is present and the fragment shader reads from an input variable decorated with `PrimitiveId`, then the geometry shader **must** write to an output variable decorated with `PrimitiveId` in all execution paths.

If a mesh shader is present and the fragment shader reads from an input variable decorated with `PrimitiveId`, then the mesh shader **must** write to the output variables decorated with `PrimitiveId` in all execution paths.

The `PrimitiveId` decoration **must** be used only within mesh, intersection, any-hit, closest hit, fragment, tessellation control, tessellation evaluation, and geometry shaders.

In an intersection, any-hit, closest hit, tessellation control, or tessellation evaluation shader, any variable decorated with `PrimitiveId` **must** be declared using the `Input` storage class.

In a geometry shader, any variable decorated with `PrimitiveId` **must** be declared using either the `Input` or `Output` storage class.

In a mesh shader, any variable decorated with `PrimitiveId` **must** be declared using the `Output` storage class.

In a fragment shader, any variable decorated with `PrimitiveId` **must** be declared using the `Input` storage class, and either the `Geometry` or `Tessellation` capability **must** also be declared.

Any variable decorated with `PrimitiveId` **must** be declared as a scalar 32-bit integer.

*Note*



When the `PrimitiveId` decoration is applied to an output variable in the mesh shader or geometry shader, the resulting value is seen through the `PrimitiveId` decorated input variable in the fragment shader.

## `PrimitiveIndicesNV`

Decorating a variable with the `PrimitiveIndicesNV` decoration will make that variable contain the output array of vertex index values. Depending on the output primitive type declared using the execution mode, the indices are split into groups of one (`OutputPoints`), two (`OutputLinesNV`), or three (`OutputTriangles`) indices and each group generates a primitive.

All index values **must** be in the range [0, N-1], where N is the value specified by the `OutputVertices` execution mode.

The `PrimitiveIndicesNV` decoration **must** only be used in mesh shaders.

Any variable decorated with `PrimitiveIndicesNV` **must** be declared using the `Output` storage class.

Any variable decorated with `PrimitiveIndicesNV` **must** be declared as an array of scalar 32-bit integers. The array **must** be sized according to the primitive type and `OutputPrimitivesNV`

execution modes, where the size is:

- the value specified by `OutputPrimitivesNV` if the execution mode is `OutputPoints`,
- two times the value specified by `OutputPrimitivesNV` if the execution mode is `OutputLinesNV`, or
- three times the value specified by `OutputPrimitivesNV` if the execution mode is `OutputTrianglesNV`.

### `RayTmaxNV`

A variable decorated with the `RayTmaxNV` decoration will contain the parametric `tmax` values of the ray being processed. The values are independent of the space in which the ray and origin exist.

The `tmax` value changes throughout the lifetime of the ray query that produced the intersection. In the closest hit shader, the value reflects the closest distance to the intersected primitive. In the any-hit shader, it reflects the distance to the primitive currently being intersected. In the intersection shader, it reflects the distance to the closest primitive intersected so far. The value can change in the intersection shader after calling `OpReportIntersectionNV` if the corresponding any-hit shader does not ignore the intersection. In a miss shader, the value is identical to the parameter passed into `OpTraceNV`.

The `RayTmaxNV` decoration **must** only be used with the intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `RayTmaxNV` **must** be declared with the `Input` storage class.

Any variable decorated with `RayTmaxNV` **must** be declared as a scalar 32-bit floating-point value.

### `RayTminNV`

A variable decorated with the `RayTminNV` decoration will contain the parametric `tmin` values of the ray being processed. The values are independent of the space in which the ray and origin exist.

The `tmin` value remains constant for the duration of the ray query.

The `RayTminNV` decoration **must** only be used with the intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `RayTminNV` **must** be declared with the `Input` storage class.

Any variable decorated with `RayTminNV` **must** be declared as a scalar 32-bit floating-point value.

### `SampleId`

Decorating a variable with the `SampleId` built-in decoration will make that variable contain the zero-based index of the sample the invocation corresponds to. `SampleId` ranges from zero to the number of samples in the framebuffer minus one. If a fragment shader entry point's interface includes an input variable decorated with `SampleId`, `Sample Shading` is considered enabled with a `minSampleShading` value of 1.0.

The `SampleId` decoration **must** be used only within fragment shaders.

The variable decorated with `SampleId` **must** be declared using the `Input` storage class.

The variable decorated with **SampleId** must be declared as a scalar 32-bit integer.

### SampleMask

Decorating a variable with the **SampleMask** built-in decoration will make any variable contain the sample coverage mask for the current fragment shader invocation.

A variable in the **Input** storage class decorated with **SampleMask** will contain a bitmask of the set of samples covered by the primitive generating the fragment during rasterization. It has a sample bit set if and only if the sample is considered covered for this fragment shader invocation. **SampleMask[]** is an array of integers. Bits are mapped to samples in a manner where bit B of mask M (**SampleMask[M]**) corresponds to sample  $32 \times M + B$ .

When state specifies multiple fragment shader invocations for a given fragment, the sample mask for any single fragment shader invocation specifies the subset of the covered samples for the fragment that correspond to the invocation. In this case, the bit corresponding to each covered sample will be set in exactly one fragment shader invocation.

If the **PostDepthCoverage** execution mode is specified, the sample is considered covered if and only if the sample is covered by the primitive and the sample passes the [early per-fragment tests](#). Otherwise the sample is considered covered if the sample is covered by the primitive, regardless of the result of the fragment tests.

A variable in the **Output** storage class decorated with **SampleMask** is an array of integers forming a bit array in a manner similar an input variable decorated with **SampleMask**, but where each bit represents coverage as computed by the shader. Modifying the sample mask by writing zero to a bit of **SampleMask** causes the sample to be considered uncovered. If this variable is also decorated with **OverrideCoverageNV**, the fragment coverage is replaced with the sample mask bits set in the shader otherwise the fragment coverage is **ANDed** with the bits of the sample mask. If the fragment shader is being evaluated at any frequency other than per-fragment, bits of the sample mask not corresponding to the current fragment shader invocation are ignored. This array **must** be sized in the fragment shader either implicitly or explicitly, to be no larger than the implementation-dependent maximum sample-mask (as an array of 32-bit elements), determined by the maximum number of samples. If a fragment shader entry point's interface includes an output variable decorated with **SampleMask**, the sample mask will be undefined for any array elements of any fragment shader invocations that fail to assign a value. If a fragment shader entry point's interface does not include an output variable decorated with **SampleMask**, the sample mask has no effect on the processing of a fragment.

The **SampleMask** decoration **must** be used only within fragment shaders.

Any variable decorated with **SampleMask** **must** be declared using either the **Input** or **Output** storage class.

Any variable decorated with **SampleMask** **must** be declared as an array of 32-bit integers.

### SamplePosition

Decorating a variable with the **SamplePosition** built-in decoration will make that variable contain the sub-pixel position of the sample being shaded. The top left of the pixel is considered to be at coordinate (0,0) and the bottom right of the pixel is considered to be at coordinate (1,1).

If the render pass has a fragment density map attachment, the variable will instead contain the sub-fragment position of the sample being shaded. The top left of the fragment is considered to be at coordinate (0,0) and the bottom right of the fragment is considered to be at coordinate (1,1) for any fragment area.

If a fragment shader entry point's interface includes an input variable decorated with `SamplePosition`, [Sample Shading](#) is considered enabled with a `minSampleShading` value of 1.0.

The `SamplePosition` decoration **must** be used only within fragment shaders.

The variable decorated with `SamplePosition` **must** be declared using the `Input` storage class. If the current pipeline uses [custom sample locations](#) the value of any variable decorated with the `SamplePosition` built-in decoration is undefined.

The variable decorated with `SamplePosition` **must** be declared as a two-component vector of 32-bit floating-point values.

#### `SMCountNV`

Decorating a variable with the `SMCountNV` built-in decoration will make that variable contain the number of SMs on the device.

The variable decorated with `SMCountNV` **must** be declared using the `Input` storage class.

The variable decorated with `SMCountNV` **must** be declared as a scalar 32-bit integer value.

#### `SMIDNV`

Decorating a variable with the `SMIDNV` built-in decoration will make that variable contain the ID of the SM on which the current shader invocation is running. This variable is in the range [0, `SMCountNV`-1].

The variable decorated with `SMIDNV` **must** be declared using the `Input` storage class.

The variable decorated with `SMIDNV` **must** be declared as a scalar 32-bit integer value.

#### `SubgroupId`

Decorating a variable with the `SubgroupId` built-in decoration will make that variable contain the index of the subgroup within the local workgroup. This variable is in range [0, `NumSubgroups`-1].

The `SubgroupId` decoration **must** be used only within task, mesh or, compute shaders.

The variable decorated with `SubgroupId` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupId` **must** be declared as a scalar 32-bit integer.

#### `SubgroupEqMask`

Decorating a variable with the `SubgroupEqMask` builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bit corresponding to the `SubgroupLocalInvocationId` is set in the variable decorated with `SubgroupEqMask`. All other bits are set to zero.

The variable decorated with `SubgroupEqMask` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupEqMask` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupEqMaskKHR` is an alias of `SubgroupEqMask`.

### `SubgroupGeMask`

Decorating a variable with the `SubgroupGeMask` builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations greater than or equal to `SubgroupLocalInvocationId` through `SubgroupSize-1` are set in the variable decorated with `SubgroupGeMask`. All other bits are set to zero.

The variable decorated with `SubgroupGeMask` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupGeMask` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupGeMaskKHR` is an alias of `SubgroupGeMask`.

### `SubgroupGtMask`

Decorating a variable with the `SubgroupGtMask` builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations greater than `SubgroupLocalInvocationId` through `SubgroupSize-1` are set in the variable decorated with `SubgroupGtMask`. All other bits are set to zero.

The variable decorated with `SubgroupGtMask` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupGtMask` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupGtMaskKHR` is an alias of `SubgroupGtMask`.

### `SubgroupLeMask`

Decorating a variable with the `SubgroupLeMask` builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations less than or equal to `SubgroupLocalInvocationId` are set in the variable decorated with `SubgroupLeMask`. All other bits are set to zero.

The variable decorated with `SubgroupLeMask` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupLeMask` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupLeMaskKHR` is an alias of `SubgroupLeMask`.

### `SubgroupLtMask`

Decorating a variable with the `SubgroupLtMask` builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations less than `SubgroupLocalInvocationId` are set in the variable decorated with `SubgroupLtMask`. All other bits are set to zero.

The variable decorated with `SubgroupLtMask` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupLtMask` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupLtMaskKHR` is an alias of `SubgroupLtMask`.

### `SubgroupLocalInvocationId`

Decorating a variable with the `SubgroupLocalInvocationId` builtin decoration will make that variable contain the index of the invocation within the subgroup. This variable is in range  $[0, \text{SubgroupSize}-1]$ .

The variable decorated with `SubgroupLocalInvocationId` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupLocalInvocationId` **must** be declared as a scalar 32-bit integer.

#### *Note*

There is no direct relationship between `SubgroupLocalInvocationId` and `LocalInvocationId` or `LocalInvocationIndex`. If the pipeline was created with `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT`, applications can compute their own local invocation index to serve the same purpose:



`index = SubgroupLocalInvocationId + SubgroupId * SubgroupSize`

If full subgroups are not enabled, some subgroups may be dispatched with inactive invocations that don't correspond to a local workgroup invocation, making the value of `index` unreliable.

### `SubgroupSize`

Decorating a variable with the `SubgroupSize` builtin decoration will make that variable contain the implementation-dependent `number of invocations in a subgroup`. This value **must** be a power-of-two integer.

If the pipeline was created with the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set, the `SubgroupSize` decorated variable will contain the subgroup size for each subgroup that gets dispatched. This value **must** be between `minSubgroupSize` and `maxSubgroupSize` and **must** be uniform with subgroup scope. The value **may** vary across a single draw or dispatch call, and for fragment shaders **may** vary across a single primitive.

If the pipeline was created with a chained `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure, the `SubgroupSize` decorated variable will match `requiredSubgroupSize`.

If the pipeline was not created with the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set and no `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure was chained, the variable decorated with `SubgroupSize` will match `subgroupSize`.

The maximum number of invocations that an implementation can support per subgroup is 128.

- + The variable decorated with **SubgroupSize** **must** be declared using the **Input** storage class.
- + The variable decorated with **SubgroupSize** **must** be declared as a scalar 32-bit integer.

### TaskCountNV

Decorating a variable with the **TaskCountNV** decoration will make that variable contain the task count. The task count specifies the number of subsequent mesh shader workgroups that get generated upon completion of the task shader.

The **TaskCountNV** decoration **must** only be used in task shaders.

Any variable decorated with **TaskCountNV** **must** be declared using the **Output** storage class.

Any variable decorated with **TaskCountNV** **must** be declared as a scalar 32-bit integer.

### TessCoord

Decorating a variable with the **TessCoord** built-in decoration will make that variable contain the three-dimensional (u,v,w) barycentric coordinate of the tessellated vertex within the patch. u, v, and w are in the range [0,1] and vary linearly across the primitive being subdivided. For the tessellation modes of **Quads** or **IsoLines**, the third component is always zero.

The **TessCoord** decoration **must** be used only within tessellation evaluation shaders.

The variable decorated with **TessCoord** **must** be declared using the **Input** storage class.

The variable decorated with **TessCoord** **must** be declared as three-component vector of 32-bit floating-point values.

### TessLevelOuter

Decorating a variable with the **TessLevelOuter** built-in decoration will make that variable contain the outer tessellation levels for the current patch.

In tessellation control shaders, the variable decorated with **TessLevelOuter** **can** be written to, which controls the tessellation factors for the resulting patch. These values are used by the tessellator to control primitive tessellation and **can** be read by tessellation evaluation shaders.

In tessellation evaluation shaders, the variable decorated with **TessLevelOuter** **can** read the values written by the tessellation control shader.

The **TessLevelOuter** decoration **must** be used only within tessellation control and tessellation evaluation shaders.

In a tessellation control shader, any variable decorated with **TessLevelOuter** **must** be declared using the **Output** storage class.

In a tessellation evaluation shader, any variable decorated with **TessLevelOuter** **must** be declared using the **Input** storage class.

Any variable decorated with **TessLevelOuter** **must** be declared as an array of size four, containing 32-bit floating-point values.

## TessLevelInner

Decorating a variable with the `TessLevelInner` built-in decoration will make that variable contain the inner tessellation levels for the current patch.

In tessellation control shaders, the variable decorated with `TessLevelInner` **can** be written to, which controls the tessellation factors for the resulting patch. These values are used by the tessellator to control primitive tessellation and **can** be read by tessellation evaluation shaders.

In tessellation evaluation shaders, the variable decorated with `TessLevelInner` **can** read the values written by the tessellation control shader.

The `TessLevelInner` decoration **must** be used only within tessellation control and tessellation evaluation shaders.

In a tessellation control shader, any variable decorated with `TessLevelInner` **must** be declared using the `Output` storage class.

In a tessellation evaluation shader, any variable decorated with `TessLevelInner` **must** be declared using the `Input` storage class.

Any variable decorated with `TessLevelInner` **must** be declared as an array of size two, containing 32-bit floating-point values.

## VertexIndex

Decorating a variable with the `VertexIndex` built-in decoration will make that variable contain the index of the vertex that is being processed by the current vertex shader invocation. For non-indexed draws, this variable begins at the `firstVertex` parameter to `vkCmdDraw` or the `firstVertex` member of a structure consumed by `vkCmdDrawIndirect` and increments by one for each vertex in the draw. For indexed draws, its value is the content of the index buffer for the vertex plus the `vertexOffset` parameter to `vkCmdDrawIndexed` or the `vertexOffset` member of the structure consumed by `vkCmdDrawIndexedIndirect`.

The `VertexIndex` decoration **must** be used only within vertex shaders.

The variable decorated with `VertexIndex` **must** be declared using the `Input` storage class.

The variable decorated with `VertexIndex` **must** be declared as a scalar 32-bit integer.



### Note

`VertexIndex` starts at the same starting value for each instance.

## ViewIndex

The `ViewIndex` decoration **can** be applied to a shader input which will be filled with the index of the view that is being processed by the current shader invocation.

If multiview is enabled in the render pass, this value will be one of the bits set in the view mask of the subpass the pipeline is compiled against. If multiview is not enabled in the render pass, this value will be zero.

The `ViewIndex` decoration **must** not be used within compute shaders.

The variable decorated with `ViewIndex` **must** be declared using the `Input` storage class.

The variable decorated with `ViewIndex` **must** be declared as a scalar 32-bit integer.

### `ViewportIndex`

Decorating a variable with the `ViewportIndex` built-in decoration will make that variable contain the index of the viewport.

In a mesh, vertex, tessellation evaluation, or geometry shader, the variable decorated with `ViewportIndex` can be written to with the viewport index to which the primitive produced by that shader will be directed.

The selected viewport index is used to select the viewport transform, scissor rectangle, and exclusive scissor rectangle.

The last active *vertex processing stage* (in pipeline order) controls the `ViewportIndex` that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the `ViewportIndex`.

If the last active vertex processing stage shader entry point's interface does not include a variable decorated with `ViewportIndex`, then the first viewport is used. If a vertex processing stage shader entry point's interface includes a variable decorated with `ViewportIndex`, it **must** write the same value to `ViewportIndex` for all output vertices of a given primitive.

The `ViewportIndex` decoration **must** be used only within mesh, vertex, tessellation evaluation, geometry, and fragment shaders.

In a mesh, vertex, tessellation evaluation, or geometry shader, any variable decorated with `ViewportIndex` **must** be declared using the `Output` storage class.

In a fragment shader, the variable decorated with `ViewportIndex` contains the viewport index of the primitive that the fragment invocation belongs to.

In a fragment shader, any variable decorated with `ViewportIndex` **must** be declared using the `Input` storage class.

Any variable decorated with `ViewportIndex` **must** be declared as a scalar 32-bit integer.

### `ViewportMaskNV`

Decorating a variable with the `ViewportMaskNV` built-in decoration will make that variable contain the viewport mask.

In a mesh, vertex, tessellation evaluation, or geometry shader, the variable decorated with `ViewportMaskNV` can be written to with the mask of which viewports the primitive produced by that shader will directed.

The `ViewportMaskNV` variable **must** be an array that has  $(\text{VkPhysicalDeviceLimits}::\text{maxViewports} / 32)$  elements. When a shader writes to this variable, bit B of element M controls whether a primitive is emitted to viewport  $32 \times M + B$ . The viewports indicated by the mask are used to select the viewport transform, scissor rectangle, and exclusive scissor rectangle that a primitive will be transformed by.

The last active *vertex processing stage* (in pipeline order) controls the `ViewportMaskNV` that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the `ViewportMaskNV`. When `ViewportMaskNV` is written by the final vertex processing stage, any variable decorated with `ViewportIndex` in the fragment shader will have the index of the viewport that was used in generating that fragment.

If a vertex processing stage shader entry point's interface includes a variable decorated with `ViewportMaskNV`, it **must** write the same value to `ViewportMaskNV` for all output vertices of a given primitive.

The `ViewportMaskNV` decoration **must** be used only within mesh, vertex, tessellation evaluation, and geometry shaders.

Any variable decorated with `ViewportMaskNV` **must** be declared using the `Output` storage class.

Any variable decorated with `ViewportMaskNV` **must** be declared as an array of 32-bit integers.

### `ViewportMaskPerViewNV`

Decorating a variable with the `ViewportMaskPerViewNV` built-in decoration will make that variable contain the mask of viewports primitives are broadcast to, for each view.

The `ViewportMaskPerViewNV` decoration **must** be used only within mesh, vertex, tessellation control, tessellation evaluation, and geometry shaders.

Any variable decorated with `ViewportMaskPerViewNV` **must** be declared using the `Output` storage class.

The value written to an element of `ViewportMaskPerViewNV` in the last vertex processing stage is a bitmask indicating which viewports the primitive will be directed to. The primitive will be broadcast to the viewport corresponding to each non-zero bit of the bitmask, and that viewport index is used to select the viewport transform, scissor rectangle, and exclusive scissor rectangle, for each view. The same values **must** be written to all vertices in a given primitive, or else the set of viewports used for that primitive is undefined.

Any variable decorated with `ViewportMaskPerViewNV` **must** be declared as an array of scalar 32-bit integers with at least as many elements as the maximum view in the subpass's view mask plus one. The array **must** be indexed by a constant or specialization constant.

Elements of the array correspond to views in a multiview subpass, and those elements corresponding to views in the view mask of the subpass the shader is compiled against will be used as the viewport mask value for those views. `ViewportMaskPerViewNV` output in an earlier vertex processing stage is not available as an input in the subsequent vertex processing stage.

Although `ViewportMaskNV` is an array, `ViewportMaskPerViewNV` is not a two-dimensional array. Instead, `ViewportMaskPerViewNV` is limited to 32 viewports.

### `WarpsPerSMNV`

Decorating a variable with the `WarpsPerSMNV` built-in decoration will make that variable contain the maximum number of warps executing on a SM.

The variable decorated with `WarpsPerSMNV` **must** be declared using the `Input` storage class.

The variable decorated with `WarpPerSMNV` **must** be declared as a scalar 32-bit integer value.

#### `WarpIDNV`

Decorating a variable with the `WarpIDNV` built-in decoration will make that variable contain the ID of the warp on a SM on which the current shader invocation is running. This variable is in the range [0, `WarpPerSMNV`-1].

The variable decorated with `WarpIDNV` **must** be declared using the `Input` storage class.

The variable decorated with `WarpIDNV` **must** be declared as a scalar 32-bit integer value.

#### `WorkgroupId`

Decorating a variable with the `WorkgroupId` built-in decoration will make that variable contain the global workgroup that the current invocation is a member of. Each component ranges from a base value to a base + count value, based on the parameters passed into the dispatch commands.

The `WorkgroupId` decoration **must** be used only within task, mesh, or compute shaders.

The variable decorated with `WorkgroupId` **must** be declared using the `Input` storage class.

The variable decorated with `WorkgroupId` **must** be declared as a three-component vector of 32-bit integers.

#### `WorkgroupSize`

Decorating an object with the `WorkgroupSize` built-in decoration will make that object contain the dimensions of a local workgroup. If an object is decorated with the `WorkgroupSize` decoration, this **must** take precedence over any execution mode set for `LocalSize`.

The `WorkgroupSize` decoration **must** be used only within task, mesh, or compute shaders.

The object decorated with `WorkgroupSize` **must** be a specialization constant or a constant.

The object decorated with `WorkgroupSize` **must** be declared as a three-component vector of 32-bit integers.

#### `WorldRayDirectionNV`

A variable decorated with the `WorldRayDirectionNV` decoration will specify the direction of the ray being processed, in world space.

The `WorldRayDirectionNV` decoration **must** only be used within intersection, any-hit, closest hit, and miss shaders.

Any variable decorated with `WorldRayDirectionNV` **must** be declared using the `Input` storage class.

Any variable decorated with `WorldRayDirectionNV` **must** be declared as a three-component vector of 32-bit floating-point values.

#### `WorldRayOriginNV`

A variable decorated with the `WorldRayOriginNV` decoration will specify the origin of the ray being processed, in world space.

The `WorldRayOriginNV` decoration **must** only be used within intersection, any-hit, closest hit, and

miss shaders.

Any variable decorated with `WorldRayOriginNV` **must** be declared using the `Input` storage class.

Any variable decorated with `WorldRayOriginNV` **must** be declared as a three-component vector of 32-bit floating-point values.

#### `WorldToObjectNV`

A variable decorated with the `WorldToObjectNV` decoration will contain the current world-to-object transformation matrix, which is determined by the instance of the current intersection.

The `WorldToObjectNV` decoration **must** only be used within intersection, any-hit, and closest hit shaders.

Any variable decorated with `WorldToObjectNV` **must** be declared using the `Input` storage class.

Any variable decorated with `WorldToObjectNV` **must** be declared as a matrix with four columns of three-component vectors of 32-bit floating-point values.

# Chapter 15. Image Operations

## 15.1. Image Operations Overview

Vulkan Image Operations are operations performed by those SPIR-V Image Instructions which take an `OpTypeImage` (representing a `VkImageView`) or `OpTypeSampledImage` (representing a (`VkImageView`, `VkSampler`) pair) and texel coordinates as operands, and return a value based on one or more neighboring texture elements (*texels*) in the image.

*Note*



Texel is a term which is a combination of the words texture and element. Early interactive computer graphics supported texture operations on textures, a small subset of the image operations on images described here. The discrete samples remain essentially equivalent, however, so we retain the historical term texel to refer to them.

Image Operations include the functionality of the following SPIR-V Image Instructions:

- `OpImageSample*` and `OpImageSparseSample*` read one or more neighboring texels of the image, and `filter` the texel values based on the state of the sampler.
  - Instructions with `ImplicitLod` in the name `determine` the LOD used in the sampling operation based on the coordinates used in neighboring fragments.
  - Instructions with `ExplicitLod` in the name `determine` the LOD used in the sampling operation based on additional coordinates.
  - Instructions with `Proj` in the name apply homogeneous `projection` to the coordinates.
- `OpImageFetch` and `OpImageSparseFetch` return a single texel of the image. No sampler is used.
- `OpImage*Gather` and `OpImageSparse*Gather` read neighboring texels and `return a single component` of each.
- `OpImageRead` (and `OpImageSparseRead`) and `OpImageWrite` read and write, respectively, a texel in the image. No sampler is used.
- `OpImageSampleFootprintNV` identifies and returns information about the set of texels in the image that would be accessed by an equivalent `OpImageSample*` instruction.
- Instructions with `Dref` in the name apply `depth comparison` on the texel values.
- Instructions with `Sparse` in the name additionally return a `sparse residency` code.

### 15.1.1. Texel Coordinate Systems

Images are addressed by *texel coordinates*. There are three *texel coordinate systems*:

- normalized texel coordinates [0.0, 1.0]
- unnormalized texel coordinates [0.0, width / height / depth)
- integer texel coordinates [0, width / height / depth)

SPIR-V `OpImageFetch`, `OpImageSparseFetch`, `OpImageRead`, `OpImageSparseRead`, and `OpImageWrite` instructions use integer texel coordinates. Other image instructions **can** use either normalized or unnormalized texel coordinates (selected by the `unnormalizedCoordinates` state of the sampler used in the instruction), but there are [limitations](#) on what operations, image state, and sampler state is supported. Normalized coordinates are logically [converted](#) to unnormalized as part of image operations, and [certain steps](#) are only performed on normalized coordinates. The array layer coordinate is always treated as unnormalized even when other coordinates are normalized.

Normalized texel coordinates are referred to as  $(s, t, r, q, a)$ , with the coordinates having the following meanings:

- s: Coordinate in the first dimension of an image.
- t: Coordinate in the second dimension of an image.
- r: Coordinate in the third dimension of an image.
  - $(s, t, r)$  are interpreted as a direction vector for Cube images.
- q: Fourth coordinate, for homogeneous (projective) coordinates.
- a: Coordinate for array layer.

The coordinates are extracted from the SPIR-V operand based on the dimensionality of the image variable and type of instruction. For `Proj` instructions, the components are in order  $(s [t] [r] q)$ , with t and r being conditionally present based on the `Dim` of the image. For non-`Proj` instructions, the coordinates are  $(s [t] [r] [a])$ , with t and r being conditionally present based on the `Dim` of the image and a being conditionally present based on the `Arrayed` property of the image. Projective image instructions are not supported on `Arrayed` images.

Unnormalized texel coordinates are referred to as  $(u, v, w, a)$ , with the coordinates having the following meanings:

- u: Coordinate in the first dimension of an image.
- v: Coordinate in the second dimension of an image.
- w: Coordinate in the third dimension of an image.
- a: Coordinate for array layer.

Only the u and v coordinates are directly extracted from the SPIR-V operand, because only 1D and 2D (non-`Arrayed`) dimensionalities support unnormalized coordinates. The components are in order  $(u [v])$ , with v being conditionally present when the dimensionality is 2D. When normalized coordinates are converted to unnormalized coordinates, all four coordinates are used.

Integer texel coordinates are referred to as  $(i, j, k, l, n)$ , with the coordinates having the following meanings:

- i: Coordinate in the first dimension of an image.
- j: Coordinate in the second dimension of an image.
- k: Coordinate in the third dimension of an image.
- l: Coordinate for array layer.

- n: Coordinate for the sample index.

They are extracted from the SPIR-V operand in order ( $i, [j], [k], [l]$ ), with j and k conditionally present based on the **Dim** of the image, and l conditionally present based on the **Arrayed** property of the image. n is conditionally present and is taken from the **Sample** image operand.

For all coordinate types, unused coordinates are assigned a value of zero.

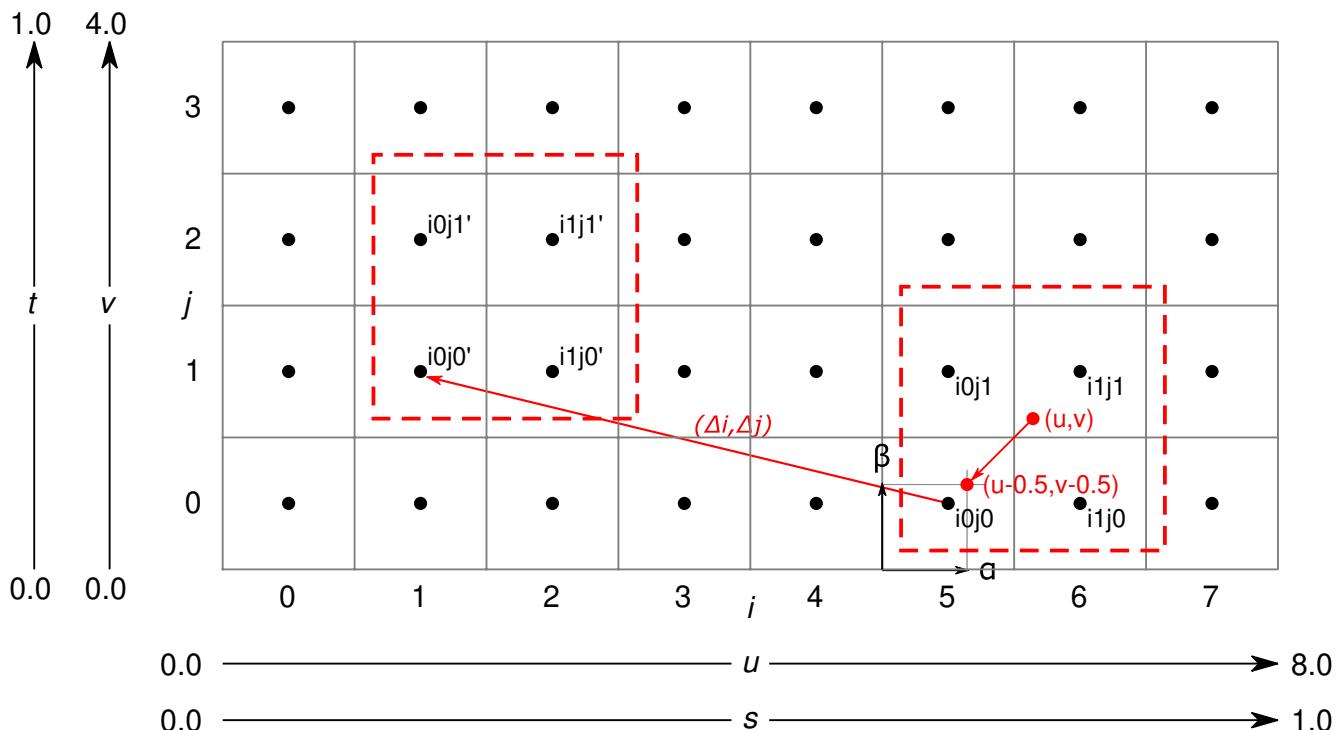


Figure 3. Texel Coordinate Systems, Linear Filtering

The Texel Coordinate Systems - For the example shown of an  $8 \times 4$  texel two dimensional image.

- Normalized texel coordinates:
  - The s coordinate goes from 0.0 to 1.0.
  - The t coordinate goes from 0.0 to 1.0.
- Unnormalized texel coordinates:
  - The u coordinate within the range 0.0 to 8.0 is within the image, otherwise it is outside the image.
  - The v coordinate within the range 0.0 to 4.0 is within the image, otherwise it is outside the image.
- Integer texel coordinates:
  - The i coordinate within the range 0 to 7 addresses texels within the image, otherwise it is outside the image.
  - The j coordinate within the range 0 to 3 addresses texels within the image, otherwise it is outside the image.
- Also shown for linear filtering:
  - Given the unnormalized coordinates  $(u, v)$ , the four texels selected are  $i_0j_0$ ,  $i_1j_0$ ,  $i_0j_1$ , and  $i_1j_1$ .

- The fractions  $\alpha$  and  $\beta$ .
- Given the offset  $\Delta_i$  and  $\Delta_j$ , the four texels selected by the offset are  $i_0j'_0$ ,  $i_1j'_0$ ,  $i_0j'_1$ , and  $i_1j'_1$ .

**Note**



For formats with reduced-resolution channels,  $\Delta_i$  and  $\Delta_j$  are relative to the resolution of the highest-resolution channel, and therefore may be divided by two relative to the unnormalized coordinate space of the lower-resolution channels.

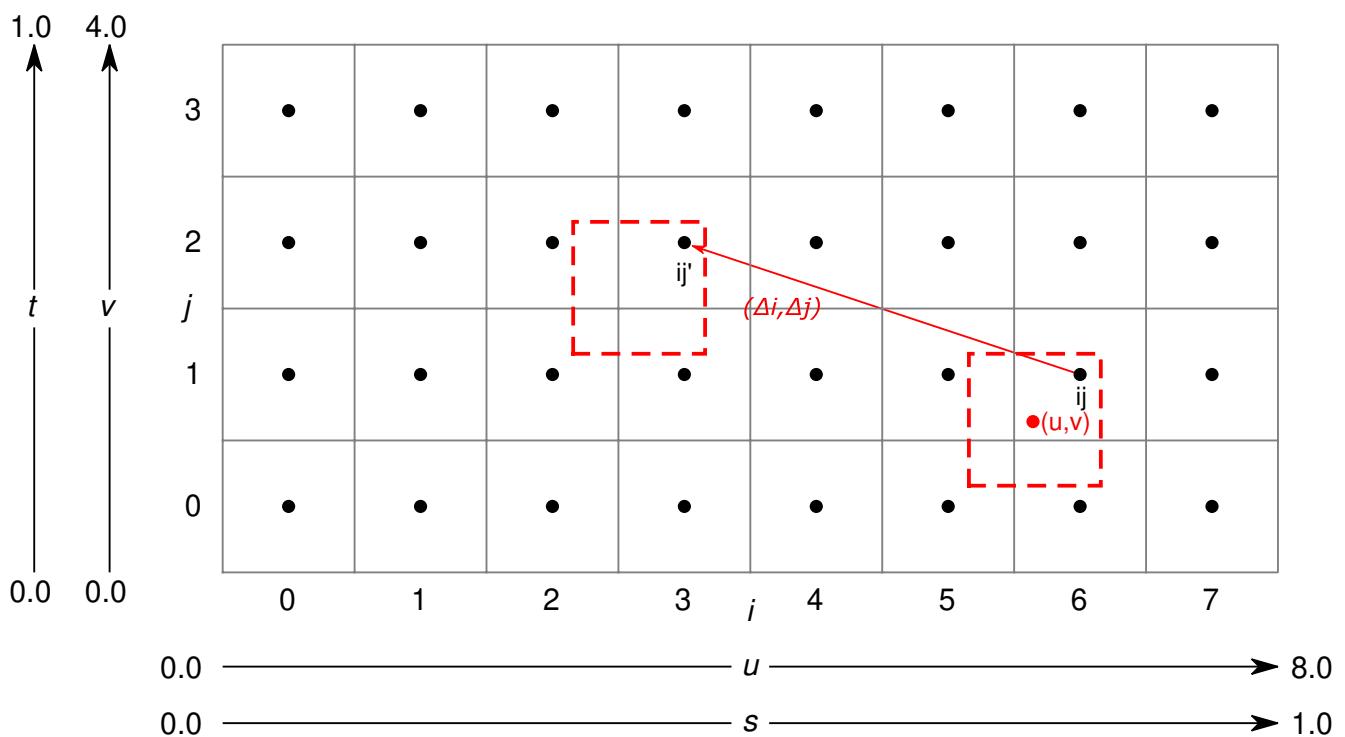


Figure 4. Texel Coordinate Systems, Nearest Filtering

The Texel Coordinate Systems - For the example shown of an  $8 \times 4$  texel two dimensional image.

- Texel coordinates as above. Also shown for nearest filtering:
  - Given the unnormalized coordinates  $(u, v)$ , the texel selected is  $ij$ .
  - Given the offset  $\Delta_i$  and  $\Delta_j$ , the texel selected by the offset is  $ij'$ .

For corner-sampled images, the texel samples are located at the grid intersections instead of the texel centers.

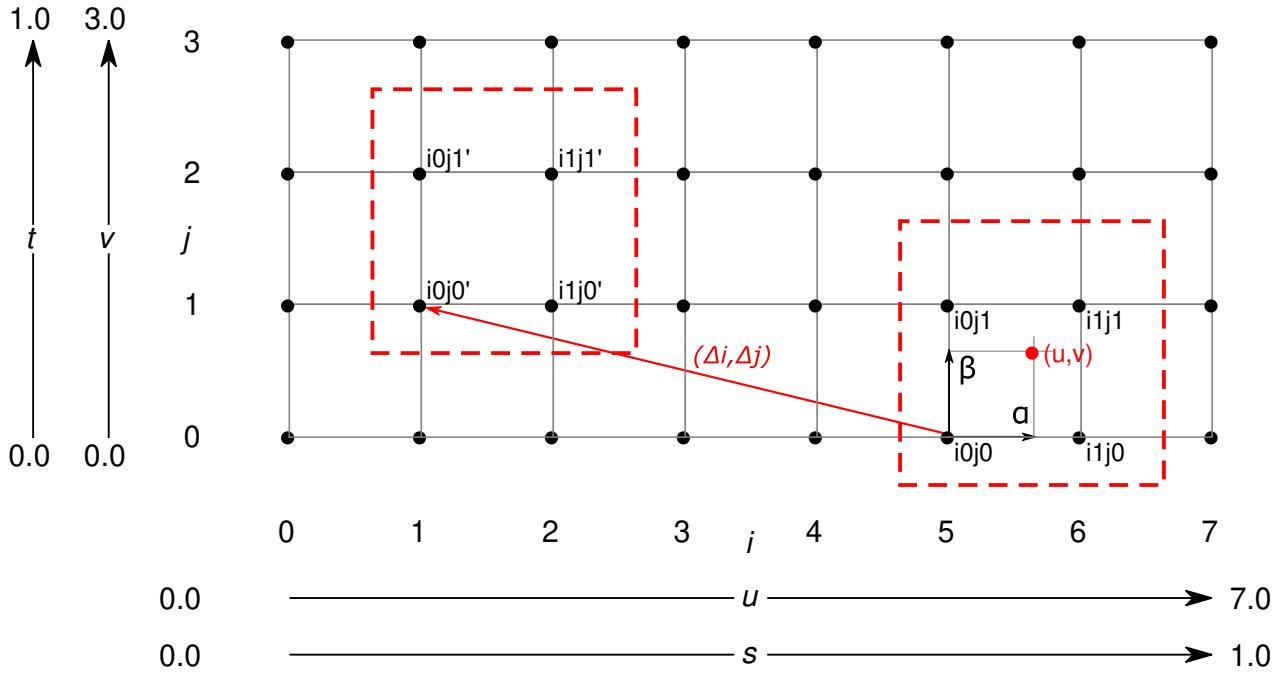


Figure 5. Texel Coordinate Systems, Corner Sampling

## 15.2. Conversion Formulas

### 15.2.1. RGB to Shared Exponent Conversion

An RGB color (red, green, blue) is transformed to a shared exponent color ( $\text{red}_{\text{shared}}$ ,  $\text{green}_{\text{shared}}$ ,  $\text{blue}_{\text{shared}}$ ,  $\text{exp}_{\text{shared}}$ ) as follows:

First, the components (red, green, blue) are clamped to ( $\text{red}_{\text{clamped}}$ ,  $\text{green}_{\text{clamped}}$ ,  $\text{blue}_{\text{clamped}}$ ) as:

$$\text{red}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{red}))$$

$$\text{green}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{green}))$$

$$\text{blue}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{blue}))$$

where:

$N = 9$	number of mantissa bits per component
$B = 15$	exponent bias
$E_{\text{max}} = 31$	maximum possible biased exponent value
$\text{sharedexp}_{\text{max}} = \frac{(2^N - 1)}{2^N} \times 2^{(E_{\text{max}} - B)}$	

Note

**i** NaN, if supported, is handled as in IEEE 754-2008 `minNum()` and `maxNum()`. That is the result is a NaN is mapped to zero.

The largest clamped component,  $\text{max}_{\text{clamped}}$  is determined:

$$\max_{\text{clamped}} = \max(\text{red}_{\text{clamped}}, \text{green}_{\text{clamped}}, \text{blue}_{\text{clamped}})$$

A preliminary shared exponent  $\exp'$  is computed:

$$\exp' = \begin{cases} \lfloor \log_2(\max_{\text{clamped}}) \rfloor + (B + 1) & \text{for } \max_{\text{clamped}} > 2^{-(B + 1)} \\ 0 & \text{for } \max_{\text{clamped}} \leq 2^{-(B + 1)} \end{cases}$$

The shared exponent  $\exp_{\text{shared}}$  is computed:

$$\max_{\text{shared}} = \lfloor \frac{\max_{\text{clamped}}}{2^{(\exp' - B - N)}} + \frac{1}{2} \rfloor$$

$$\exp_{\text{shared}} = \begin{cases} \exp' & \text{for } 0 \leq \max_{\text{shared}} < 2^N \\ \exp' + 1 & \text{for } \max_{\text{shared}} = 2^N \end{cases}$$

Finally, three integer values in the range 0 to  $2^N$  are computed:

$$\begin{aligned} \text{red}_{\text{shared}} &= \lfloor \frac{\text{red}_{\text{clamped}}}{2^{(\exp_{\text{shared}} - B - N)}} + \frac{1}{2} \rfloor \\ \text{green}_{\text{shared}} &= \lfloor \frac{\text{green}_{\text{clamped}}}{2^{(\exp_{\text{shared}} - B - N)}} + \frac{1}{2} \rfloor \\ \text{blue}_{\text{shared}} &= \lfloor \frac{\text{blue}_{\text{clamped}}}{2^{(\exp_{\text{shared}} - B - N)}} + \frac{1}{2} \rfloor \end{aligned}$$

### 15.2.2. Shared Exponent to RGB

A shared exponent color ( $\text{red}_{\text{shared}}$ ,  $\text{green}_{\text{shared}}$ ,  $\text{blue}_{\text{shared}}$ ,  $\exp_{\text{shared}}$ ) is transformed to an RGB color (red, green, blue) as follows:

$$\text{red} = \text{red}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)}$$

$$\text{green} = \text{green}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)}$$

$$\text{blue} = \text{blue}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)}$$

where:

$$N = 9 \text{ (number of mantissa bits per component)}$$

$$B = 15 \text{ (exponent bias)}$$

## 15.3. Texel Input Operations

*Texel input instructions* are SPIR-V image instructions that read from an image. *Texel input operations* are a set of steps that are performed on state, coordinates, and texel values while processing a texel input instruction, and which are common to some or all texel input instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Instruction/Sampler/Image validation
  - Coordinate validation
  - Sparse validation
  - Layout validation
- Format conversion
- Texel replacement
- Depth comparison
- Conversion to RGBA
- Component swizzle
- Chroma reconstruction
- Y'C<sub>B</sub>C<sub>R</sub> conversion

For texel input instructions involving multiple texels (for sampling or gathering), these steps are applied for each texel that is used in the instruction. Depending on the type of image instruction, other steps are conditionally performed between these steps or involving multiple coordinate or texel values.

If [Chroma Reconstruction](#) is implicit, [Texel Filtering](#) instead takes place during chroma reconstruction, before sampler Y'C<sub>B</sub>C<sub>R</sub> conversion occurs.

### 15.3.1. Texel Input Validation Operations

*Texel input validation operations* inspect instruction/image/sampler state or coordinates, and in certain circumstances cause the texel value to be replaced or become undefined. There are a series of validations that the texel undergoes.

#### Instruction/Sampler/Image View Validation

There are a number of cases where a SPIR-V instruction **can** mismatch with the sampler, the image view, or both. There are a number of cases where the sampler **can** mismatch with the image view. In such cases the value of the texel returned is undefined.

These cases include:

- The sampler `borderColor` is an integer type and the image view `format` is not one of the `VkFormat` integer types or a stencil component of a depth/stencil format.
- The sampler `borderColor` is a float type and the image view `format` is not one of the `VkFormat` float types or a depth component of a depth/stencil format.
- The sampler `borderColor` is one of the opaque black colors (`VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK` or `VK_BORDER_COLOR_INT_OPAQUE_BLACK`) and the image view `VkComponentSwizzle` for any of the `VkComponentMapping` components is not `VK_COMPONENT_SWIZZLE_IDENTITY`.
- The `VkImageLayout` of any subresource in the image view does not match that specified in `VkDescriptorImageInfo::imageLayout` used to write the image descriptor.

- If the instruction is `OpImageRead` or `OpImageSparseRead` and the `shaderStorageImageReadWithoutFormat` feature is not enabled, or the instruction is `OpImageWrite` and the `shaderStorageImageWriteWithoutFormat` feature is not enabled, then the SPIR-V Image Format **must** be compatible with the image view's `format`.
- The sampler `unnormalizedCoordinates` is `VK_TRUE` and any of the limitations of unnormalized coordinates are violated.
- The sampler was created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` and the image was not created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.
- The sampler was not created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` and the image was created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.
- The sampler was created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` and is used with a function that is not `OpImageSampleImplicitLod` or `OpImageSampleExplicitLod`, or is used with operands `Offset` or `ConstOffsets`.
- The SPIR-V instruction is one of the `OpImage*Dref*` instructions and the sampler `compareEnable` is `VK_FALSE`
- The SPIR-V instruction is not one of the `OpImage*Dref*` instructions and the sampler `compareEnable` is `VK_TRUE`
- The SPIR-V instruction is one of the `OpImage*Dref*` instructions and the image view `format` is not one of the depth/stencil formats with a depth component, or the image view aspect is not `VK_IMAGE_ASPECT_DEPTH_BIT`.
- The SPIR-V instruction's image variable's properties are not compatible with the image view:
  - Rules for `viewType`:
    - `VK_IMAGE_VIEW_TYPE_1D` **must** have `Dim` = 1D, `Arrayed` = 0, `MS` = 0.
    - `VK_IMAGE_VIEW_TYPE_2D` **must** have `Dim` = 2D, `Arrayed` = 0.
    - `VK_IMAGE_VIEW_TYPE_3D` **must** have `Dim` = 3D, `Arrayed` = 0, `MS` = 0.
    - `VK_IMAGE_VIEW_TYPE_CUBE` **must** have `Dim` = Cube, `Arrayed` = 0, `MS` = 0.
    - `VK_IMAGE_VIEW_TYPE_1D_ARRAY` **must** have `Dim` = 1D, `Arrayed` = 1, `MS` = 0.
    - `VK_IMAGE_VIEW_TYPE_2D_ARRAY` **must** have `Dim` = 2D, `Arrayed` = 1.
    - `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY` **must** have `Dim` = Cube, `Arrayed` = 1, `MS` = 0.
  - If the image was created with `VkImageCreateInfo::samples` equal to `VK_SAMPLE_COUNT_1_BIT`, the instruction **must** have `MS` = 0.
  - If the image was created with `VkImageCreateInfo::samples` not equal to `VK_SAMPLE_COUNT_1_BIT`, the instruction **must** have `MS` = 1.
- If the image was created with `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV`, the sampler addressing modes **must** only use a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- The SPIR-V instruction is `OpImageSampleFootprintNV` with `Dim` = 2D and `addressModeU` or `addressModeV` in the sampler is not `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- The SPIR-V instruction is `OpImageSampleFootprintNV` with `Dim` = 3D and `addressModeU`, `addressModeV`, or `addressModeW` in the sampler is not `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

Only `OpImageSample*` and `OpImageSparseSample*` can be used with a sampler that enables sampler  $\text{Y}'\text{C}_B\text{C}_R$  conversion.

`OpImageFetch`, `OpImageSparseFetch`, `OpImage*Gather`, and `OpImageSparse*Gather` must not be used with a sampler that enables sampler  $\text{Y}'\text{C}_B\text{C}_R$  conversion.

The `ConstOffset` and `Offset` operands must not be used with a sampler that enables sampler  $\text{Y}'\text{C}_B\text{C}_R$  conversion.

## Integer Texel Coordinate Validation

Integer texel coordinates are validated against the size of the image level, and the number of layers and number of samples in the image. For SPIR-V instructions that use integer texel coordinates, this is performed directly on the integer coordinates. For instructions that use normalized or unnormalized texel coordinates, this is performed on the coordinates that result after conversion to integer texel coordinates.

If the integer texel coordinates do not satisfy all of the conditions

$$0 \leq i < w_s$$

$$0 \leq j < h_s$$

$$0 \leq k < d_s$$

$$0 \leq l < \text{layers}$$

$$0 \leq n < \text{samples}$$

where:

$w_s$  = width of the image level

$h_s$  = height of the image level

$d_s$  = depth of the image level

$\text{layers}$  = number of layers in the image

$\text{samples}$  = number of samples per texel in the image

then the texel fails integer texel coordinate validation.

There are four cases to consider:

### 1. Valid Texel Coordinates

- If the texel coordinates pass validation (that is, the coordinates lie within the image), then the texel value comes from the value in image memory.

## 2. Border Texel

- If the texel coordinates fail validation, and
- If the read is the result of an image sample instruction or image gather instruction, and
- If the image is not a cube image,

then the texel is a border texel and [texel replacement](#) is performed.

## 3. Invalid Texel

- If the texel coordinates fail validation, and
- If the read is the result of an image fetch instruction, image read instruction, or atomic instruction,

then the texel is an invalid texel and [texel replacement](#) is performed.

## 4. Cube Map Edge or Corner

Otherwise the texel coordinates lie beyond the edges or corners of the selected cube map face, and [Cube map edge handling](#) is performed.

### Cube Map Edge Handling

If the texel coordinates lie beyond the edges or corners of the selected cube map face, the following steps are performed. Note that this does not occur when using `VK_FILTER_NEAREST` filtering within a mip level, since `VK_FILTER_NEAREST` is treated as using `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

- Cube Map Edge Texel

- If the texel lies beyond the selected cube map face in either only i or only j, then the coordinates (i,j) and the array layer l are transformed to select the adjacent texel from the appropriate neighboring face.

- Cube Map Corner Texel

- If the texel lies beyond the selected cube map face in both i and j, then there is no unique neighboring face from which to read that texel. The texel **should** be replaced by the average of the three values of the adjacent texels in each incident face. However, implementations **may** replace the cube map corner texel by other methods. The methods are subject to the constraint that for linear filtering if the three available texels have the same value, the resulting filtered texel **must** have that value, and for cubic filtering if the twelve available samples have the same value, the resulting filtered texel **must** have that value.

### Sparse Validation

If the texel reads from an unbound region of a sparse image, the texel is a *sparse unbound texel*, and processing continues with [texel replacement](#).

### Layout Validation

If all planes of a *disjoint multi-planar* image are not in the same [image layout](#), the image **must** not be sampled with [sampler Y'C<sub>B</sub>C<sub>R</sub> conversion](#) enabled.

### 15.3.2. Format Conversion

Texels undergo a format conversion from the [VkFormat](#) of the image view to a vector of either floating point or signed or unsigned integer components, with the number of components based on the number of components present in the format.

- Color formats have one, two, three, or four components, according to the format.
- Depth/stencil formats are one component. The depth or stencil component is selected by the [aspectMask](#) of the image view.

Each component is converted based on its type and size (as defined in the [Format Definition](#) section for each [VkFormat](#)), using the appropriate equations in [16-Bit Floating-Point Numbers](#), [Unsigned 11-Bit Floating-Point Numbers](#), [Unsigned 10-Bit Floating-Point Numbers](#), [Fixed-Point Data Conversion](#), and [Shared Exponent to RGB](#). Signed integer components smaller than 32 bits are sign-extended.

If the image view format is sRGB, the color components are first converted as if they are UNORM, and then sRGB to linear conversion is applied to the R, G, and B components as described in the “sRGB EOTF” section of the [Khronos Data Format Specification](#). The A component, if present, is unchanged.

If the image view format is block-compressed, then the texel value is first decoded, then converted based on the type and number of components defined by the compressed format.

### 15.3.3. Texel Replacement

A texel is replaced if it is one (and only one) of:

- a border texel,
- an invalid texel, or
- a sparse unbound texel.

Border texels are replaced with a value based on the image format and the [borderColor](#) of the sampler. The border color is:

*Table 22. Border Color B*

Sampler <a href="#">borderColor</a>	Corresponding Border Color
<a href="#">VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK</a>	$[B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 0.0]$
<a href="#">VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK</a>	$[B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 1.0]$
<a href="#">VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE</a>	$[B_r, B_g, B_b, B_a] = [1.0, 1.0, 1.0, 1.0]$
<a href="#">VK_BORDER_COLOR_INT_TRANSPARENT_BLACK</a>	$[B_r, B_g, B_b, B_a] = [0, 0, 0, 0]$
<a href="#">VK_BORDER_COLOR_INT_OPAQUE_BLACK</a>	$[B_r, B_g, B_b, B_a] = [0, 0, 0, 1]$
<a href="#">VK_BORDER_COLOR_INT_OPAQUE_WHITE</a>	$[B_r, B_g, B_b, B_a] = [1, 1, 1, 1]$

*Note*



The names `VK_BORDER_COLOR_*_TRANSPARENT_BLACK`, `VK_BORDER_COLOR_*_OPAQUE_BLACK`, and `VK_BORDER_COLOR_*_OPAQUE_WHITE` are meant to describe which components are zeros and ones in the vocabulary of compositing, and are not meant to imply that the numerical value of `VK_BORDER_COLOR_INT_OPAQUE_WHITE` is a saturating value for integers.

This is substituted for the texel value by replacing the number of components in the image format

*Table 23. Border Texel Components After Replacement*

Texel Aspect or Format	Component Assignment
Depth aspect	$D = B_r$
Stencil aspect	$S = B_r$
One component color format	$\text{Color}_r = B_r$
Two component color format	$[\text{Color}_r, \text{Color}_g] = [B_r, B_g]$
Three component color format	$[\text{Color}_r, \text{Color}_g, \text{Color}_b] = [B_r, B_g, B_b]$
Four component color format	$[\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [B_r, B_g, B_b, B_a]$

The value returned by a read of an invalid texel is undefined, unless that read operation is from a buffer resource and the `robustBufferAccess` feature is enabled. In that case, an invalid texel is replaced as described by the `robustBufferAccess` feature.

If the `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict` property is `VK_TRUE`, a sparse unbound texel is replaced with 0 or 0.0 values for integer and floating-point components of the image format, respectively.

If `residencyNonResidentStrict` is `VK_FALSE`, the value of the sparse unbound texel is undefined.

### 15.3.4. Depth Compare Operation

If the image view has a depth/stencil format, the depth component is selected by the `aspectMask`, and the operation is a `Dref` instruction, a depth comparison is performed. The value of the result  $D$  is 1.0 if the result of the compare operation is true, and 0.0 otherwise. The compare operation is selected by the `compareOp` member of the sampler.

$$D = \begin{cases} 1.0 & \begin{cases} D_{ref} \leq D & \text{for LEQUAL} \\ D_{ref} \geq D & \text{for GEQUAL} \\ D_{ref} < D & \text{for LESS} \\ D_{ref} > D & \text{for GREATER} \\ D_{ref} = D & \text{for EQUAL} \\ D_{ref} \neq D & \text{for NOTEQUAL} \\ \text{true} & \text{for ALWAYS} \\ \text{false} & \text{for NEVER} \end{cases} \\ 0.0 & \text{otherwise} \end{cases}$$

where, in the depth comparison:

$D_{ref}$  = shaderOp.D<sub>ref</sub> (from **optional** SPIR-V operand)

D (texel depth value)

### 15.3.5. Conversion to RGBA

The texel is expanded from one, two, or three components to four components based on the image base color:

Table 24. Texel Color After Conversion To RGBA

Texel Aspect or Format	RGBA Color
Depth aspect	[Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , Color <sub>a</sub> ] = [D, 0, 0, one]
Stencil aspect	[Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , Color <sub>a</sub> ] = [S, 0, 0, one]
One component color format	[Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , Color <sub>a</sub> ] = [Color <sub>r</sub> , 0, 0, one]
Two component color format	[Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , Color <sub>a</sub> ] = [Color <sub>r</sub> , Color <sub>g</sub> , 0, one]
Three component color format	[Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , Color <sub>a</sub> ] = [Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , one]
Four component color format	[Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , Color <sub>a</sub> ] = [Color <sub>r</sub> , Color <sub>g</sub> , Color <sub>b</sub> , Color <sub>a</sub> ]

where one = 1.0f for floating-point formats and depth aspects, and one = 1 for integer formats and stencil aspects.

### 15.3.6. Component Swizzle

All texel input instructions apply a *swizzle* based on:

- the `VkComponentSwizzle` enums in the `components` member of the `VkImageViewCreateInfo` structure for the image being read if `sampler Y'CBCR conversion` is not enabled, and
- the `VkComponentSwizzle` enums in the `components` member of the `VkSamplerYcbcrConversionCreateInfo` structure for the `sampler Y'CBCR conversion` if `sampler Y'CBCR conversion` is enabled.

The swizzle **can** rearrange the components of the texel, or substitute zero or one for any components. It is defined as follows for each color component:

$$Color'_{component} = \begin{cases} Color_r & \text{for RED swizzle} \\ Color_g & \text{for GREEN swizzle} \\ Color_b & \text{for BLUE swizzle} \\ Color_a & \text{for ALPHA swizzle} \\ 0 & \text{for ZERO swizzle} \\ one & \text{for ONE swizzle} \\ identity & \text{for IDENTITY swizzle} \end{cases}$$

where:

$$one = \begin{cases} 1.0f & \text{for floating point components} \\ 1 & \text{for integer components} \end{cases}$$

$$identity = \begin{cases} Color_r & \text{for component} = r \\ Color_g & \text{for component} = g \\ Color_b & \text{for component} = b \\ Color_a & \text{for component} = a \end{cases}$$

If the border color is one of the `VK_BORDER_COLOR_*_OPAQUE_BLACK` enums and the `VkComponentSwizzle` is not `VK_COMPONENT_SWIZZLE_IDENTITY` for all components (or the equivalent identity mapping), the value of the texel after swizzle is undefined.

### 15.3.7. Sparse Residency

`OpImageSparse*` instructions return a structure which includes a *residency code* indicating whether any texels accessed by the instruction are sparse unbound texels. This code **can** be interpreted by the `OpImageSparseTexelsResident` instruction which converts the residency code to a boolean value.

### 15.3.8. Chroma Reconstruction

In some color models, the color representation is defined in terms of monochromatic light intensity (often called “luma”) and color differences relative to this intensity, often called “chroma”. It is common for color models other than RGB to represent the chroma channels at lower spatial resolution than the luma channel. This approach is used to take advantage of the eye’s lower spatial sensitivity to color compared with its sensitivity to brightness. Less commonly, the same approach is used with additive color, since the green channel dominates the eye’s sensitivity to light intensity and the spatial sensitivity to color introduced by red and blue is lower.

Lower-resolution channels are “downsampled” by resizing them to a lower spatial resolution than the channel representing luminance. The process of reconstructing a full color value for texture access involves accessing both chroma and luma values at the same location. To generate the color accurately, the values of the lower-resolution channels at the location of the luma samples must be reconstructed from the lower-resolution sample locations, an operation known here as “chroma reconstruction” irrespective of the actual color model.

The location of the chroma samples relative to the luma coordinates is determined by the `xChromaOffset` and `yChromaOffset` members of the `VkSamplerYcbcrConversionCreateInfo` structure used to create the sampler Y’C<sub>B</sub>C<sub>R</sub> conversion.

The following diagrams show the relationship between unnormalized  $(u,v)$  coordinates and  $(i,j)$  integer texel positions in the luma channel (shown in black, with circles showing integer sample positions) and the texel coordinates of reduced-resolution chroma channels, shown as crosses in red.

**Note**



If the chroma values are reconstructed at the locations of the luma samples by means of interpolation, chroma samples from outside the image bounds are needed; these are determined according to [Wrapping Operation](#). These diagrams represent this by showing the bounds of the “chroma texel” extending beyond the image bounds, and including additional chroma sample positions where required for interpolation. The limits of a sample for **NEAREST** sampling is shown as a grid.

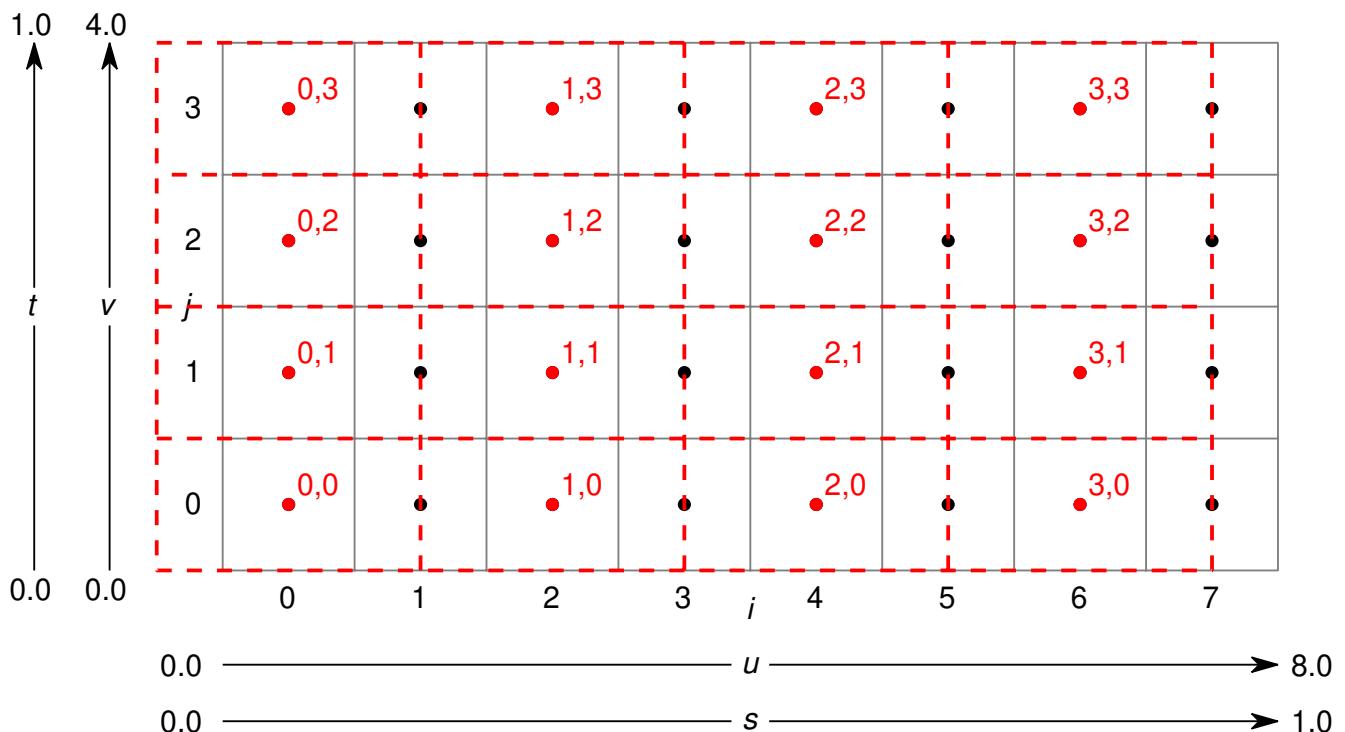


Figure 6. 422 downsampling,  $xChromaOffset=COSITED\_EVEN$

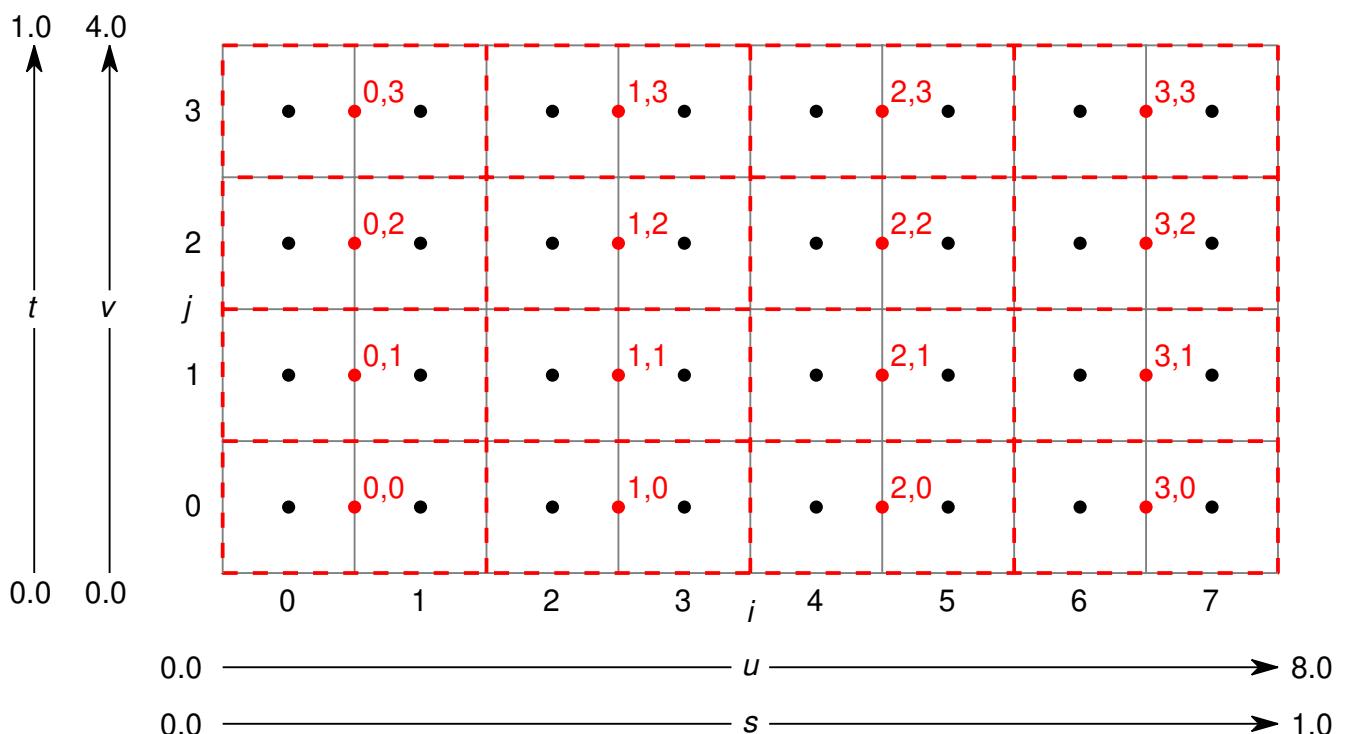


Figure 7. 422 downsampling,  $xChromaOffset=MIDPOINT$

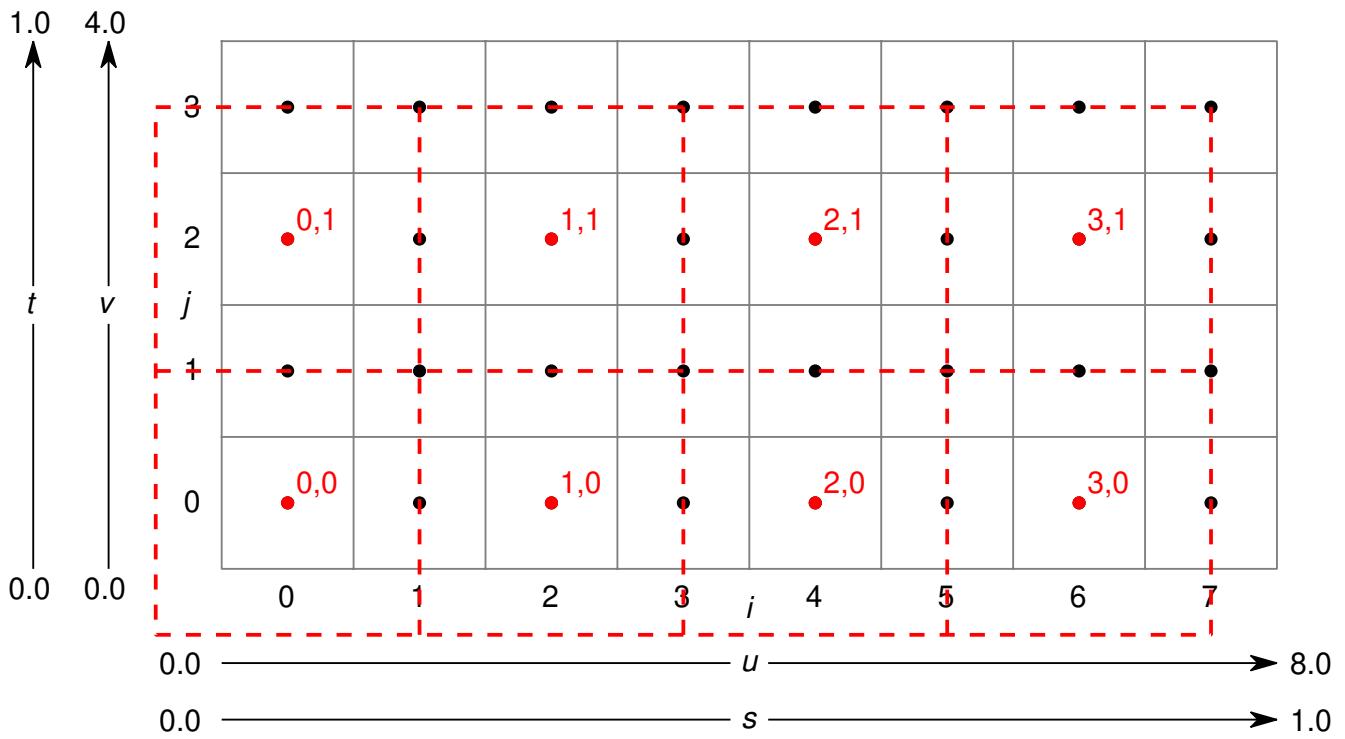


Figure 8. 420 downsampling,  $xChromaOffset=COSITED\_EVEN$ ,  $yChromaOffset=COSITED\_EVEN$

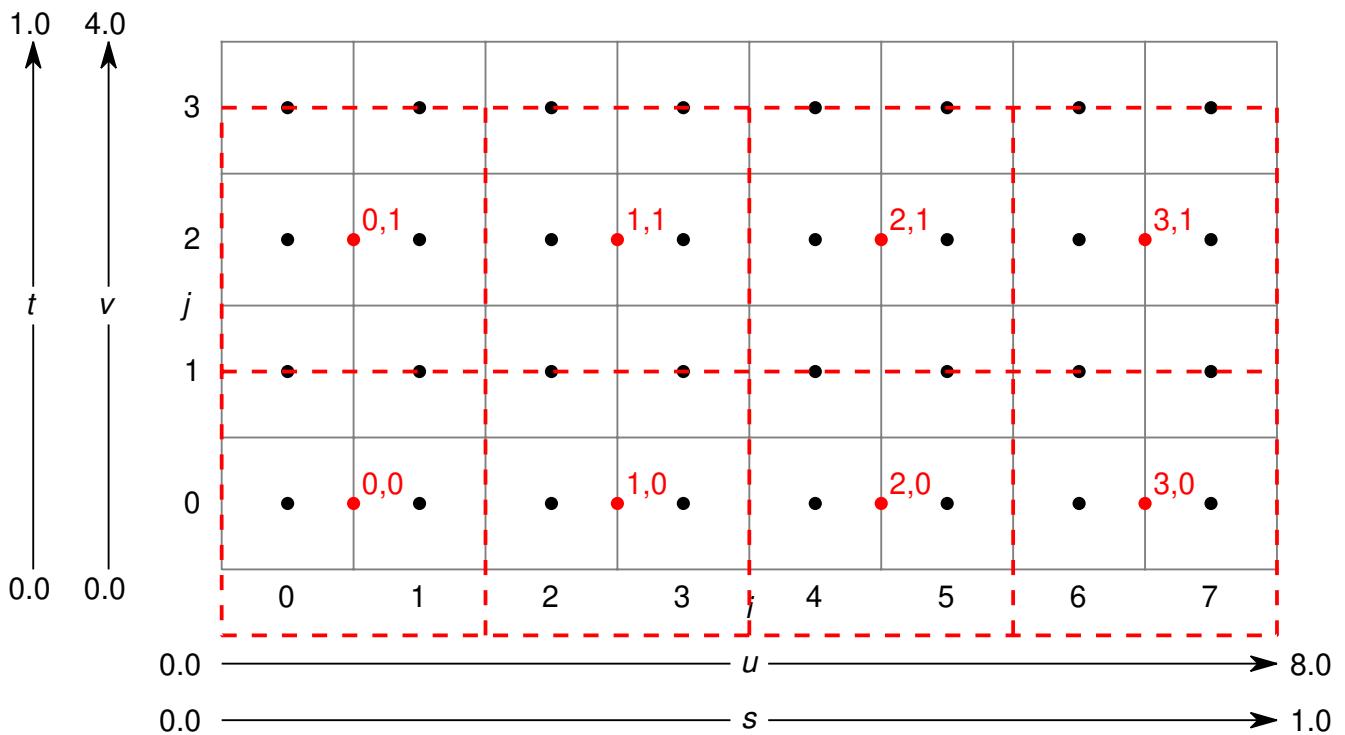


Figure 9. 420 downsampling,  $xChromaOffset=MIDPOINT$ ,  $yChromaOffset=COSITED\_EVEN$

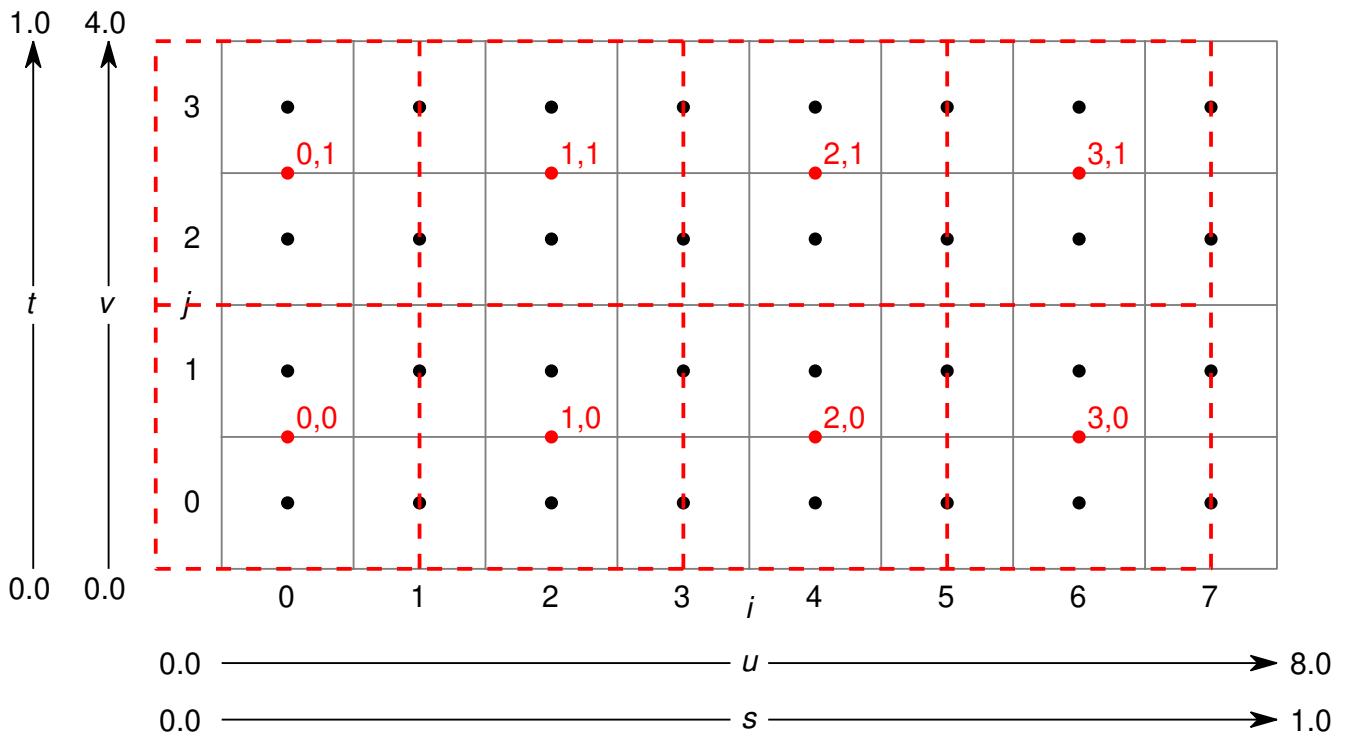


Figure 10. 420 downsampling, `xChromaOffset=COSITED_EVEN`, `yChromaOffset=MIDPOINT`

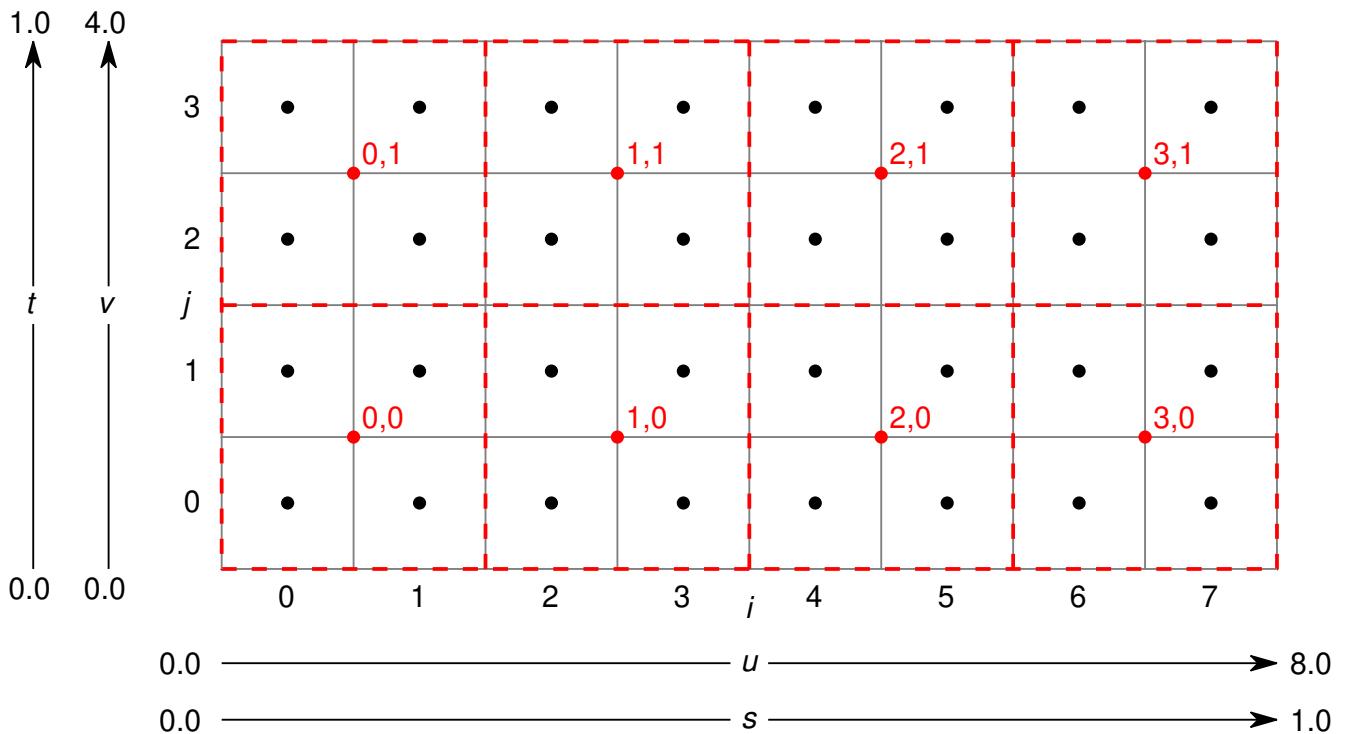


Figure 11. 420 downsampling, `xChromaOffset=MIDPOINT`, `yChromaOffset=MIDPOINT`

Reconstruction is implemented in one of two ways:

If the format of the image that is to be sampled sets `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT`, or the `VkSamplerYcbcrConversionCreateInfo`'s `forceExplicitReconstruction` is set to `VK_TRUE`, reconstruction is performed as an explicit step independent of filtering, described in the [Explicit Reconstruction](#) section.

If the format of the image that is to be sampled does not set `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT` and if the `VkSamplerYcbcrConversionCreateInfo`'s `forceExplicitReconstruction` is set to `VK_FALSE`, reconstruction is performed as an implicit part of filtering prior to color model conversion, with no separate post-conversion texel filtering step, as described in the [Implicit Reconstruction](#) section.

## Explicit Reconstruction

- If the `chromaFilter` member of the `VkSamplerYcbcrConversionCreateInfo` structure is `VK_FILTER_NEAREST`:
  - If the format's R and B channels are reduced in resolution in just width by a factor of two relative to the G channel (i.e. this is a “`_422`” format), the  $\tau_{ijk}[level]$  values accessed by `texel filtering` are reconstructed as follows:

$$\begin{aligned}\tau_R'(i, j) &= \tau_R(\lfloor i \times 0.5 \rfloor, j)[level] \\ \tau_B'(i, j) &= \tau_B(\lfloor i \times 0.5 \rfloor, j)[level]\end{aligned}$$

- If the format's R and B channels are reduced in resolution in width and height by a factor of two relative to the G channel (i.e. this is a “`_420`” format), the  $\tau_{ijk}[level]$  values accessed by `texel filtering` are reconstructed as follows:

$$\begin{aligned}\tau_R'(i, j) &= \tau_R(\lfloor i \times 0.5 \rfloor, \lfloor j \times 0.5 \rfloor)[level] \\ \tau_B'(i, j) &= \tau_B(\lfloor i \times 0.5 \rfloor, \lfloor j \times 0.5 \rfloor)[level]\end{aligned}$$

*Note*



`xChromaOffset` and `yChromaOffset` have no effect if `chromaFilter` is `VK_FILTER_NEAREST` for explicit reconstruction.

- If the `chromaFilter` member of the `VkSamplerYcbcrConversionCreateInfo` structure is `VK_FILTER_LINEAR`:
  - If the format's R and B channels are reduced in resolution in just width by a factor of two relative to the G channel (i.e. this is a “`422`” format):
    - If `xChromaOffset` is `VK_CHROMA_LOCATION_COSITED_EVEN`:

$$\tau_{RB}'(i, j) = \begin{cases} \tau_{RB}(\lfloor i \times 0.5 \rfloor, j)[level], & 0.5 \times i = \lfloor 0.5 \times i \rfloor \\ 0.5 \times \tau_{RB}(\lfloor i \times 0.5 \rfloor, j)[level] + \\ 0.5 \times \tau_{RB}(\lfloor i \times 0.5 \rfloor + 1, j)[level], & 0.5 \times i \neq \lfloor 0.5 \times i \rfloor \end{cases}$$

- If `xChromaOffset` is `VK_CHROMA_LOCATION_MIDPOINT`:

$$\tau_{RB}(i, j)' = \begin{cases} 0.25 \times \tau_{RB}(\lfloor i \times 0.5 \rfloor - 1, j)[level] + \\ 0.75 \times \tau_{RB}(\lfloor i \times 0.5 \rfloor, j)[level], & 0.5 \times i = \lfloor 0.5 \times i \rfloor \\ 0.75 \times \tau_{RB}(\lfloor i \times 0.5 \rfloor, j)[level] + \\ 0.25 \times \tau_{RB}(\lfloor i \times 0.5 \rfloor + 1, j)[level], & 0.5 \times i \neq \lfloor 0.5 \times i \rfloor \end{cases}$$

- If the format's R and B channels are reduced in resolution in width and height by a factor of two relative to the G channel (i.e. this is a “`420`” format), a similar relationship applies. Due to the number of options, these formulae are expressed more concisely as follows:

$$i_{RB} = \begin{cases} 0.5 \times (i) & \text{If} \text{fxChromaOffset}=\text{COSITED\_EVEN} \\ 0.5 \times (i - 0.5) & \text{If} \text{fxChromaOffset}=\text{MIDPOINT} \end{cases}$$

$$j_{RB} = \begin{cases} 0.5 \times (j) & \text{If} \text{fyChromaOffset}=\text{COSITED\_EVEN} \\ 0.5 \times (j - 0.5) & \text{If} \text{fyChromaOffset}=\text{MIDPOINT} \end{cases}$$

$$i_{floor} = \lfloor i_{RB} \rfloor$$

$$j_{floor} = \lfloor j_{RB} \rfloor$$

$$i_{frac} = i_{RB} - i_{floor}$$

$$j_{frac} = j_{RB} - j_{floor}$$

$$\begin{aligned} \tau_{RB}'(i, j) &= \tau_{RB}(i_{floor}, j_{floor})[level] && \times (1 - i_{frac}) && \times (1 - j_{frac}) + \\ &\quad \tau_{RB}(1 + i_{floor}, j_{floor})[level] && \times (i_{frac}) && \times (1 - j_{frac}) + \\ &\quad \tau_{RB}(i_{floor}, 1 + j_{floor})[level] && \times (1 - i_{frac}) && \times (j_{frac}) + \\ &\quad \tau_{RB}(1 + i_{floor}, 1 + j_{floor})[level] && \times (i_{frac}) && \times (j_{frac}) \end{aligned}$$

#### Note

In the case where the texture itself is bilinearly interpolated as described in [Texel Filtering](#), thus requiring four full-color samples for the filtering operation, and where the reconstruction of these samples uses bilinear interpolation in the chroma channels due to `chromaFilter=VK_FILTER_LINEAR`, up to nine chroma samples may be required, depending on the sample location.



### Implicit Reconstruction

Implicit reconstruction takes place by the samples being interpolated, as required by the filter settings of the sampler, except that `chromaFilter` takes precedence for the chroma samples.

If `chromaFilter` is `VK_FILTER_NEAREST`, an implementation **may** behave as if `xChromaOffset` and `yChromaOffset` were both `VK_CHROMA_LOCATION_MIDPOINT`, irrespective of the values set.

#### Note



This will not have any visible effect if the locations of the luma samples coincide with the location of the samples used for rasterization.

The sample coordinates are adjusted by the downsample factor of the channel (such that, for example, the sample coordinates are divided by two if the channel has a downsample factor of two relative to the luma channel):

$$u_{RB}'(422 / 420) = \begin{cases} 0.5 \times (u + 0.5), & \text{xChromaOffset}=\text{COSITED\_EVEN} \\ 0.5 \times u, & \text{xChromaOffset}=\text{MIDPOINT} \end{cases}$$

$$v_{RB}'(420) = \begin{cases} 0.5 \times (v + 0.5), & \text{yChromaOffset}=\text{COSITED\_EVEN} \\ 0.5 \times v, & \text{yChromaOffset}=\text{MIDPOINT} \end{cases}$$

### 15.3.9. Sampler Y'CBC<sub>R</sub> Conversion

Sampler Y'CBC<sub>R</sub> conversion performs the following operations, which an implementation **may** combine into a single mathematical operation:

- Sampler Y'CBC<sub>R</sub> Range Expansion

- Sampler Y'CbCr Model Conversion

## Sampler Y'CbCr Range Expansion

Sampler Y'CbCr range expansion is applied to color channel values after all texel input operations which are not specific to sampler Y'CbCr conversion. For example, the input values to this stage have been converted using the normal [format conversion](#) rules.

Sampler Y'CbCr range expansion is not applied if `ycbcrModel` is `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`. That is, the shader receives the vector  $C'_{rgba}$  as output by the Component Swizzle stage without further modification.

For other values of `ycbcrModel`, range expansion is applied to the texel channel values output by the [Component Swizzle](#) defined by the `components` member of [`VkSamplerYcbcrConversionCreateInfo`](#). Range expansion applies independently to each channel of the image. For the purposes of range expansion and Y'CbCr model conversion, the R and B channels contain color difference (chroma) values and the G channel contains luma. The A channel is not modified by sampler Y'CbCr range expansion.

The range expansion to be applied is defined by the `ycbcrRange` member of the [`VkSamplerYcbcrConversionCreateInfo`](#) structure:

- If `ycbcrRange` is `VK_SAMPLER_YCBCR_RANGE_ITU_FULL`, the following transformations are applied:

$$\begin{aligned} Y' &= C'_{rgba}[G] \\ C_B &= C'_{rgba}[B] - \frac{2^{(n-1)}}{(2^n)-1} \\ C_R &= C'_{rgba}[R] - \frac{2^{(n-1)}}{(2^n)-1} \end{aligned}$$

### Note

These formulae correspond to the “full range” encoding in the [Khronos Data Format Specification](#).



Should any future amendments be made to the ITU specifications from which these equations are derived, the formulae used by Vulkan **may** also be updated to maintain parity.

- If `ycbcrRange` is `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW`, the following transformations are applied:

$$\begin{aligned} Y' &= \frac{C'_{rgba}[G] \times (2^n-1) - 16 \times 2^n - 8}{219 \times 2^{n-8}} \\ C_B &= \frac{C'_{rgba}[B] \times (2^n-1) - 128 \times 2^{n-8}}{224 \times 2^{n-8}} \\ C_R &= \frac{C'_{rgba}[R] \times (2^n-1) - 128 \times 2^{n-8}}{224 \times 2^{n-8}} \end{aligned}$$

### Note



These formulae correspond to the “narrow range” encoding in the [Khronos Data Format Specification](#).

- $n$  is the bit-depth of the channels in the format.

The precision of the operations performed during range expansion **must** be at least that of the source format.

An implementation **may** clamp the results of these range expansion operations such that  $Y'$  falls in the range [0,1], and/or such that  $C_B$  and  $C_R$  fall in the range [-0.5,0.5].

### **Sampler $Y'C_BC_R$ Model Conversion**

The range-expanded values are converted between color models, according to the color model conversion specified in the `ycbcrModel` member:

#### **`VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`**

The color channels are not modified by the color model conversion since they are assumed already to represent the desired color model in which the shader is operating;  $Y'C_BC_R$  range expansion is also ignored.

#### **`VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY`**

The color channels are not modified by the color model conversion and are assumed to be treated as though in  $Y'C_BC_R$  form both in memory and in the shader;  $Y'C_BC_R$  range expansion is applied to the channels as for other  $Y'C_BC_R$  models, with the vector ( $C_R, Y', C_B, A$ ) provided to the shader.

#### **`VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709`**

The color channels are transformed from a  $Y'C_BC_R$  representation to an  $R'G'B'$  representation as described in the “BT.709  $Y'C_BC_R$  conversion” section of the [Khronos Data Format Specification](#).

#### **`VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601`**

The color channels are transformed from a  $Y'C_BC_R$  representation to an  $R'G'B'$  representation as described in the “BT.601  $Y'C_BC_R$  conversion” section of the [Khronos Data Format Specification](#).

#### **`VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020`**

The color channels are transformed from a  $Y'C_BC_R$  representation to an  $R'G'B'$  representation as described in the “BT.2020  $Y'C_BC_R$  conversion” section of the [Khronos Data Format Specification](#).

In this operation, each output channel is dependent on each input channel.

An implementation **may** clamp the  $R'G'B'$  results of these conversions to the range [0,1].

The precision of the operations performed during model conversion **must** be at least that of the source format.

The alpha channel is not modified by these model conversions.

#### Note

Sampling operations in a non-linear color space can introduce color and intensity shifts at sharp transition boundaries. To avoid this issue, the technically precise color correction sequence described in the “Introduction to Color Conversions” chapter of the [Khronos Data Format Specification](#) may be performed as follows:

- Calculate the [unnormalized texel coordinates](#) corresponding to the desired sample position.
- For a [minFilter/magFilter](#) of [VK\\_FILTER\\_NEAREST](#):
  1. Calculate  $(i,j)$  for the sample location as described under the “nearest filtering” formulae in [\(u,v,w,a\) to \(i,j,k,l,n\) Transformation And Array Layer Selection](#)
  2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
  3. Sample using [sampler Y'CbCr conversion](#) at this location.
- For a [minFilter/magFilter](#) of [VK\\_FILTER\\_LINEAR](#):
  1. Calculate  $(i_{[0,1]},j_{[0,1]})$  for the sample location as described under the “linear filtering” formulae in [\(u,v,w,a\) to \(i,j,k,l,n\) Transformation And Array Layer Selection](#)
  2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
  3. Sample using [sampler Y'CbCr conversion](#) at each of these locations.
  4. Convert the non-linear AR'G'B' outputs of the Y'CbCr conversions to linear ARGB values as described in the “Transfer Functions” chapter of the [Khronos Data Format Specification](#).
  5. Interpolate the linear ARGB values using the  $\alpha$  and  $\beta$  values described in the “linear filtering” section of [\(u,v,w,a\) to \(i,j,k,l,n\) Transformation And Array Layer Selection](#) and the equations in [Texel Filtering](#).



The additional calculations and, especially, additional number of sampling operations in the [VK\\_FILTER\\_LINEAR](#) case can be expected to have a performance impact compared with using the outputs directly; since the variation from “correct” results are subtle for most content, the application author should determine whether a more costly implementation is strictly necessary. Note that if [chromaFilter](#) and [minFilter/magFilter](#) are both [VK\\_FILTER\\_NEAREST](#), these operations are redundant and sampling using [sampler Y'CbCr conversion](#) at the desired sample coordinates will produce the “correct” results without further processing.

## 15.4. Texel Output Operations

*Texel output instructions* are SPIR-V image instructions that write to an image. *Texel output operations* are a set of steps that are performed on state, coordinates, and texel values while processing a texel output instruction, and which are common to some or all texel output

instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Format validation
  - Coordinate validation
  - Sparse validation
- Texel output format conversion

### 15.4.1. Texel Output Validation Operations

*Texel output validation operations* inspect instruction/image state or coordinates, and in certain circumstances cause the write to have no effect. There are a series of validations that the texel undergoes.

#### Texel Format Validation

If the image format of the [OpTypeImage](#) is not compatible with the [VkImageView](#)'s [format](#), the write causes the contents of the image's memory to become undefined.

#### 15.4.2. Integer Texel Coordinate Validation

The integer texel coordinates are validated according to the same rules as for texel input [coordinate validation](#).

If the texel fails integer texel coordinate validation, then the write has no effect.

#### 15.4.3. Sparse Texel Operation

If the texel attempts to write to an unbound region of a sparse image, the texel is a sparse unbound texel. In such a case, if the [VkPhysicalDeviceSparseProperties::residencyNonResidentStrict](#) property is [VK\\_TRUE](#), the sparse unbound texel write has no effect. If [residencyNonResidentStrict](#) is [VK\\_FALSE](#), the write **may** have a side effect that becomes visible to other accesses to unbound texels in any resource, but will not be visible to any device memory allocated by the application.

#### 15.4.4. Texel Output Format Conversion

If the image format is sRGB, a linear to sRGB conversion is applied to the R, G, and B components as described in the “sRGB EOTF” section of the [Kronos Data Format Specification](#). The A component, if present, is unchanged.

Texels then undergo a format conversion from the floating point, signed, or unsigned integer type of the texel data to the [VkFormat](#) of the image view. Any unused components are ignored.

Each component is converted based on its type and size (as defined in the [Format Definition](#) section for each [VkFormat](#)). Floating-point outputs are converted as described in [Floating-Point Format Conversions](#) and [Fixed-Point Data Conversion](#). Integer outputs are converted such that their value is preserved. The converted value of any integer that cannot be represented in the target format is undefined.

## 15.5. Derivative Operations

SPIR-V derivative instructions include `OpDPdx`, `OpDPdy`, `OpDPdxFine`, `OpDPdyFine`, `OpDPdxCoarse`, and `OpDPdyCoarse`. Derivative instructions are only available in compute and fragment shaders.

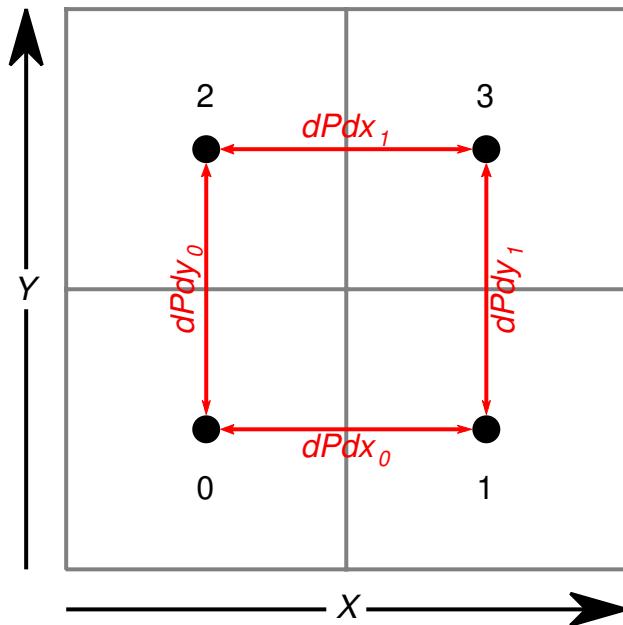


Figure 12. Implicit Derivatives

Derivatives are computed as if there is a  $2 \times 2$  neighborhood of fragments for each fragment shader invocation. These neighboring fragments are used to compute derivatives with the assumption that the values of P in the neighborhood are piecewise linear. It is further assumed that the values of P in the neighborhood are locally continuous. Applications **must** not use derivative instructions in non-uniform control flow.

$$\begin{aligned}dPdx_0 &= P_{i_1, j_0} - P_{i_0, j_0} \\dPdx_1 &= P_{i_1, j_1} - P_{i_0, j_1} \\dPdy_0 &= P_{i_0, j_1} - P_{i_0, j_0} \\dPdy_1 &= P_{i_1, j_1} - P_{i_1, j_0}\end{aligned}$$

For a  $2 \times 2$  neighborhood, for the four fragments labeled 0, 1, 2 and 3, the **Fine** derivative instructions **must** return:

$$\begin{aligned}dPdx &= \begin{cases} dPdx_0 & \text{for fragments labeled 0 and 1} \\ dPdx_1 & \text{for fragments labeled 2 and 3} \end{cases} \\dPdy &= \begin{cases} dPdy_0 & \text{for fragments labeled 0 and 2} \\ dPdy_1 & \text{for fragments labeled 1 and 3} \end{cases}\end{aligned}$$

Coarse derivatives **may** return only two values. In this case, the values **should** be:

$$\begin{aligned}dPdx &= \begin{cases} dPdx_0 & \text{preferred} \\ dPdx_1 \end{cases} \\dPdy &= \begin{cases} dPdy_0 & \text{preferred} \\ dPdy_1 \end{cases}\end{aligned}$$

`OpDPdx` and `OpDPdy` **must** return the same result as either `OpDPdxFine` or `OpDPdxCoarse` and either

`OpDPdyFine` or `OpDPdyCoarse`, respectively. Implementations **must** make the same choice of either coarse or fine for both `OpDPdx` and `OpDPdy`, and implementations **should** make the choice that is more efficient to compute.

If the `subgroupSize` field of `VkPhysicalDeviceSubgroupProperties` is at least 4, the  $2 \times 2$  neighborhood of fragments corresponds exactly to a subgroup quad. The order in which the fragments appear within the quad is implementation defined.

### 15.5.1. Compute Shader Derivatives

For compute shaders, derivatives are also evaluated using a  $2 \times 2$  logical neighborhood of compute shader invocations. Compute shader invocations are arranged into neighborhoods according to one of two SPIR-V execution modes. For the `DerivativeGroupQuadsNV` execution mode, each neighborhood is assembled from a  $2 \times 2 \times 1$  region of invocations based on the `LocalInvocationId` built-in. For the `DerivativeGroupLinearNV` execution mode, each neighborhood is assembled from a group of four invocations based on the `LocalInvocationIndex` built-in. The [Compute shader derivative group assignments](#) table specifies the `LocalInvocationId` or `LocalInvocationIndex` values for the four values of P in each neighborhood, where x and y are per-neighborhood integer values.

*Table 25. Compute shader derivative group assignments*

Value	<code>DerivativeGroupQuadsNV</code>	<code>DerivativeGroupLinearNV</code>
$P_{i0,j0}$	$(2x + 0, 2y + 0, z)$	$4x + 0$
$P_{i1,j0}$	$(2x + 1, 2y + 0, z)$	$4x + 1$
$P_{i0,j1}$	$(2x + 0, 2y + 1, z)$	$4x + 2$
$P_{i1,j1}$	$(2x + 1, 2y + 1, z)$	$4x + 3$

For multi-planar formats, the derivatives are computed based on the plane with the largest dimensions.

## 15.6. Normalized Texel Coordinate Operations

If the image sampler instruction provides normalized texel coordinates ( $s, t, r, q, a$ ) and (if present) the  $D_{ref}$  coordinate are transformed as follows:

$$\begin{aligned} s &= \frac{s}{q}, && \text{for 1D, 2D, or 3D image} \\ t &= \frac{t}{q}, && \text{for 2D or 3D image} \\ r &= \frac{r}{q}, && \text{for 3D image} \\ D_{ref} &= \frac{D_{ref}}{q}, && \text{if provided} \end{aligned}$$

## 15.6.2. Derivative Image Operations

Derivatives are used for LOD selection. These derivatives are either implicit (in an `ImplicitLod` image instruction in a fragment shader) or explicit (provided explicitly by shader to the image instruction in any shader).

For implicit derivatives image instructions, the derivatives of texel coordinates are calculated in the same manner as derivative operations above. That is:

$$\begin{array}{ll} \frac{\partial s}{\partial x} = dPdx(s), & \frac{\partial s}{\partial y} = dPdy(s), \\ \frac{\partial t}{\partial x} = dPdx(t), & \frac{\partial t}{\partial y} = dPdy(t), \\ \frac{\partial u}{\partial x} = dPdx(u), & \frac{\partial u}{\partial y} = dPdy(u), \end{array} \begin{array}{l} \text{for 1D, 2D, Cube, or 3D image} \\ \text{for 2D, Cube, or 3D image} \\ \text{for Cube or 3D image} \end{array}$$

Partial derivatives not defined above for certain image dimensionalities are set to zero.

For explicit LOD image instructions, if the **optional** SPIR-V operand `Grad` is provided, then the operand values are used for the derivatives. The number of components present in each derivative for a given image dimensionality matches the number of partial derivatives computed above.

If the **optional** SPIR-V operand `Lod` is provided, then derivatives are set to zero, the cube map derivative transformation is skipped, and the scale factor operation is skipped. Instead, the floating point scalar coordinate is directly assigned to  $\lambda_{base}$  as described in [Level-of-Detail Operation](#).

For implicit derivative image instructions, the partial derivative values **may** be computed by linear approximation using a  $2 \times 2$  neighborhood of shader invocations (known as a *quad*), as described above. If the instruction is in control flow that is not uniform across the quad, then the derivative values and hence the implicit LOD values are undefined.

If the image or sampler object used by an implicit derivative image instruction is not uniform across the quad and `quadDivergentImplicitLod` is not supported, then the derivative and LOD values are undefined. Implicit derivatives are well-defined when the image and sampler and control flow are uniform across the quad, even if they diverge between different quads.

If `quadDivergentImplicitLod` is supported, then derivatives and implicit LOD values are well-defined even if the image or sampler object are not uniform within a quad. The derivatives are computed as specified above, and the implicit LOD calculation proceeds for each shader invocation using its respective image and sampler object.

For the purposes of implicit derivatives, `Flat` fragment input variables are uniform within a quad.

## 15.6.3. Cube Map Face Selection and Transformations

For cube map image instructions, the  $(s, t, r)$  coordinates are treated as a direction vector  $(r_x, r_y, r_z)$ . The direction vector is used to select a cube map face. The direction vector is transformed to a per-face texel coordinate system  $(s_{face}, t_{face})$ . The direction vector is also used to transform the derivatives to per-face derivatives.

## 15.6.4. Cube Map Face Selection

The direction vector selects one of the cube map's faces based on the largest magnitude coordinate direction (the major axis direction). Since two or more coordinates **can** have identical magnitude,

the implementation **must** have rules to disambiguate this situation.

The rules **should** have as the first rule that  $r_z$  wins over  $r_y$  and  $r_x$ , and the second rule that  $r_y$  wins over  $r_x$ . An implementation **may** choose other rules, but the rules **must** be deterministic and depend only on  $(r_x, r_y, r_z)$ .

The layer number (corresponding to a cube map face), the coordinate selections for  $s_c$ ,  $t_c$ ,  $r_c$ , and the selection of derivatives, are determined by the major axis direction as specified in the following two tables.

*Table 26. Cube map face and coordinate selection*

Major Axis Direction	Layer Number	Cube Map Face	$s_c$	$t_c$	$r_c$
$+r_x$	0	Positive X	$-r_z$	$-r_y$	$r_x$
$-r_x$	1	Negative X	$+r_z$	$-r_y$	$r_x$
$+r_y$	2	Positive Y	$+r_x$	$+r_z$	$r_y$
$-r_y$	3	Negative Y	$+r_x$	$-r_z$	$r_y$
$+r_z$	4	Positive Z	$+r_x$	$-r_y$	$r_z$
$-r_z$	5	Negative Z	$-r_x$	$-r_y$	$r_z$

*Table 27. Cube map derivative selection*

Major Axis Direction	$\partial s_c / \partial x$	$\partial s_c / \partial y$	$\partial t_c / \partial x$	$\partial t_c / \partial y$	$\partial r_c / \partial x$	$\partial r_c / \partial y$
$+r_x$	$-\partial r_z / \partial x$	$-\partial r_z / \partial y$	$-\partial r_y / \partial x$	$-\partial r_y / \partial y$	$+\partial r_x / \partial x$	$+\partial r_x / \partial y$
$-r_x$	$+\partial r_z / \partial x$	$+\partial r_z / \partial y$	$-\partial r_y / \partial x$	$-\partial r_y / \partial y$	$-\partial r_x / \partial x$	$-\partial r_x / \partial y$
$+r_y$	$+\partial r_x / \partial x$	$+\partial r_x / \partial y$	$+\partial r_z / \partial x$	$+\partial r_z / \partial y$	$+\partial r_y / \partial x$	$+\partial r_y / \partial y$
$-r_y$	$+\partial r_x / \partial x$	$+\partial r_x / \partial y$	$-\partial r_z / \partial x$	$-\partial r_z / \partial y$	$-\partial r_y / \partial x$	$-\partial r_y / \partial y$
$+r_z$	$+\partial r_x / \partial x$	$+\partial r_x / \partial y$	$-\partial r_y / \partial x$	$-\partial r_y / \partial y$	$+\partial r_z / \partial x$	$+\partial r_z / \partial y$
$-r_z$	$-\partial r_x / \partial x$	$-\partial r_x / \partial y$	$-\partial r_y / \partial x$	$-\partial r_y / \partial y$	$-\partial r_z / \partial x$	$-\partial r_z / \partial y$

## 15.6.5. Cube Map Coordinate Transformation

$$s_{face} = \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2}$$

$$t_{face} = \frac{1}{2} \times \frac{t_c}{|r_c|} + \frac{1}{2}$$

## 15.6.6. Cube Map Derivative Transformation

$$\frac{\partial s_{face}}{\partial x} = \frac{\partial}{\partial x} \left( \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2} \right)$$

$$\frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \frac{\partial}{\partial x} \left( \frac{s_c}{|r_c|} \right)$$

$$\frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial s_c / \partial x - s_c \times \partial r_c / \partial x}{(r_c)^2} \right)$$

$$\frac{\partial s_{face}}{\partial y} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial s_c / \partial y - s_c \times \partial r_c / \partial y}{(r_c)^2} \right)$$

$$\frac{\partial t_{face}}{\partial x} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial t_c / \partial x - t_c \times \partial r_c / \partial x}{(r_c)^2} \right)$$

$$\frac{\partial t_{face}}{\partial y} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial t_c / \partial y - t_c \times \partial r_c / \partial y}{(r_c)^2} \right)$$

### 15.6.7. Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection

LOD selection **can** be either explicit (provided explicitly by the image instruction) or implicit (determined from a scale factor calculated from the derivatives). The implicit LOD selected **can** be queried using the SPIR-V instruction `OpImageQueryLod`, which gives access to the  $\lambda'$  and  $d_l$  values, defined below. These values **must** be computed with `mipmapPrecisionBits` of accuracy and **may** be subject to implementation-specific maxima and minima for very large, out-of-range values.

#### Scale Factor Operation

The magnitude of the derivatives are calculated by:

$$m_{ux} = |\partial s/\partial x| \times w_{base}$$

$$m_{vx} = |\partial t/\partial x| \times h_{base}$$

$$m_{wx} = |\partial r/\partial x| \times d_{base}$$

$$m_{uy} = |\partial s/\partial y| \times w_{base}$$

$$m_{vy} = |\partial t/\partial y| \times h_{base}$$

$$m_{wy} = |\partial r/\partial y| \times d_{base}$$

where:

$$\partial t/\partial x = \partial t/\partial y = 0 \text{ (for 1D images)}$$

$$\partial r/\partial x = \partial r/\partial y = 0 \text{ (for 1D, 2D or Cube images)}$$

and:

$$w_{base} = \text{image.w}$$

$h_{base} = \text{image}.h$

$d_{base} = \text{image}.d$

(for the `baseMipLevel`, from the image descriptor).

For corner-sampled images, the  $w_{base}$ ,  $h_{base}$ , and  $d_{base}$  are instead:

$w_{base} = \text{image}.w - 1$

$h_{base} = \text{image}.h - 1$

$d_{base} = \text{image}.d - 1$

A point sampled in screen space has an elliptical footprint in texture space. The minimum and maximum scale factors ( $\rho_{min}$ ,  $\rho_{max}$ ) **should** be the minor and major axes of this ellipse.

The *scale factors*  $\rho_x$  and  $\rho_y$ , calculated from the magnitude of the derivatives in x and y, are used to compute the minimum and maximum scale factors.

$\rho_x$  and  $\rho_y$  **may** be approximated with functions  $f_x$  and  $f_y$ , subject to the following constraints:

$f_x$  is continuous and monotonically increasing in each of  $m_{ux}$ ,  $m_{vx}$ , and  $m_{wx}$   
 $f_y$  is continuous and monotonically increasing in each of  $m_{uy}$ ,  $m_{vy}$ , and  $m_{wy}$

$$\begin{aligned} \max(|m_{ux}|, |m_{vx}|, |m_{wx}|) &\leq f_x \leq \sqrt{2}(|m_{ux}| + |m_{vx}| + |m_{wx}|) \\ \max(|m_{uy}|, |m_{vy}|, |m_{wy}|) &\leq f_y \leq \sqrt{2}(|m_{uy}| + |m_{vy}| + |m_{wy}|) \end{aligned}$$

The minimum and maximum scale factors ( $\rho_{min}$ ,  $\rho_{max}$ ) are determined by:

$\rho_{max} = \max(\rho_x, \rho_y)$

$\rho_{min} = \min(\rho_x, \rho_y)$

The ratio of anisotropy is determined by:

$\eta = \min(\rho_{max}/\rho_{min}, \text{max}_{\text{Aniso}})$

where:

`sampler.maxAniso` = `maxAnisotropy` (from sampler descriptor)

`limits.maxAniso` = `maxSamplerAnisotropy` (from physical device limits)

$\text{max}_{\text{Aniso}} = \min(\text{sampler}.max_{\text{Aniso}}, \text{limits}.max_{\text{Aniso}})$

If  $\rho_{max} = \rho_{min} = 0$ , then all the partial derivatives are zero, the fragment's footprint in texel space is a point, and  $N$  **should** be treated as 1. If  $\rho_{max} \neq 0$  and  $\rho_{min} = 0$  then all partial derivatives along one axis are zero, the fragment's footprint in texel space is a line segment, and  $\eta$  **should** be treated as

$\max_{\text{Aniso}}$ . However, anytime the footprint is small in texel space the implementation **may** use a smaller value of  $\eta$ , even when  $\rho_{\min}$  is zero or close to zero. If either `VkPhysicalDeviceFeatures::samplerAnisotropy` or `VkSamplerCreateInfo::anisotropyEnable` are `VK_FALSE`,  $\max_{\text{Aniso}}$  is set to 1.

If  $\eta = 1$ , sampling is isotropic. If  $\eta > 1$ , sampling is anisotropic.

The sampling rate ( $N$ ) is derived as:

$$N = \eta$$

An implementation **may** round  $N$  up to the nearest supported sampling rate. An implementation **may** use the value of  $N$  as an approximation of  $\eta$ .

## Level-of-Detail Operation

The LOD parameter  $\lambda$  is computed as follows:

$$\begin{aligned} \lambda_{\text{base}}(x, y) &= \begin{cases} \text{shaderOp.Lod} & (\text{from optional SPIR-V operand}) \\ \log_2\left(\frac{\rho_{\max}}{\eta}\right) & \text{otherwise} \end{cases} \\ \lambda'(x, y) &= \lambda_{\text{base}} + \text{clamp}(\text{sampler.bias} + \text{shaderOp.bias}, -\max_{\text{SamplerLodBias}}, \max_{\text{SamplerLodBias}}) \\ \lambda &= \begin{cases} \text{lod}_{\max}, & \lambda' > \text{lod}_{\max} \\ \lambda', & \text{lod}_{\min} \leq \lambda' \leq \text{lod}_{\max} \\ \text{lod}_{\min}, & \lambda' < \text{lod}_{\min} \\ \text{undefined}, & \text{lod}_{\min} > \text{lod}_{\max} \end{cases} \end{aligned}$$

where:

$$\begin{aligned} \text{sampler.bias} &= \text{mipLodBias} && (\text{from sampler descriptor}) \\ \text{shaderOp.bias} &= \begin{cases} \text{Bias} & (\text{from optional SPIR-V operand}) \\ 0 & \text{otherwise} \end{cases} \\ \text{sampler.lod}_{\min} &= \text{minLod} && (\text{from sampler descriptor}) \\ \text{shaderOp.lod}_{\min} &= \begin{cases} \text{MinLod} & (\text{from optional SPIR-V operand}) \\ 0 & \text{otherwise} \end{cases} \\ \text{lod}_{\min} &= \max(\text{sampler.lod}_{\min}, \text{shaderOp.lod}_{\min}) \\ \text{lod}_{\max} &= \text{maxLod} && (\text{from sampler descriptor}) \end{aligned}$$

and  $\max_{\text{SamplerLodBias}}$  is the value of the `VkPhysicalDeviceLimits` feature `maxSamplerLodBias`.

## Image Level(s) Selection

The image level(s)  $d$ ,  $d_{\text{hi}}$ , and  $d_{\text{lo}}$  which texels are read from are determined by an image-level parameter  $d_l$ , which is computed based on the LOD parameter, as follows:

$$d_l = \begin{cases} \text{nearest}(d'), & \text{mipmapMode is VK_SAMPLER_MIPMAP_MODE_NEAREST} \\ d', & \text{otherwise} \end{cases}$$

where:

$$d' = \text{level}_{\text{base}} + \text{clamp}(\lambda, 0, q)$$

$$\text{nearest}(d') = \begin{cases} \lceil d' + 0.5 \rceil - 1, & \text{preferred} \\ \lfloor d' + 0.5 \rfloor, & \text{alternative} \end{cases}$$

and:

```
levelbase = baseMipLevel
```

```
q = levelCount - 1
```

baseMipLevel and levelCount are taken from the subresourceRange of the image view.

If the sampler's mipmapMode is VK\_SAMPLER\_MIPMAP\_MODE\_NEAREST, then the level selected is  $d = d_l$ .

If the sampler's mipmapMode is VK\_SAMPLER\_MIPMAP\_MODE\_LINEAR, two neighboring levels are selected:

$$\begin{aligned}d_{hi} &= \lfloor d_l \rfloor \\d_{lo} &= \min(d_{hi} + 1, q) \\d &= d_l - d_{hi}\end{aligned}$$

$\delta$  is the fractional value, quantized to the number of mipmap precision bits, used for linear filtering between levels.

### 15.6.8. (s,t,r,q,a) to (u,v,w,a) Transformation

The normalized texel coordinates are scaled by the image level dimensions and the array layer is selected.

This transformation is performed once for each level used in filtering (either  $d$ , or  $d_{hi}$  and  $d_{lo}$ ).

$$\begin{aligned}u(x, y) &= s(x, y) \times width_{scale} + \Delta_i \\v(x, y) &= \begin{cases} 0 & \text{for 1D images} \\ t(x, y) \times height_{scale} + \Delta_j & \text{otherwise} \end{cases} \\w(x, y) &= \begin{cases} 0 & \text{for 2D or Cube images} \\ r(x, y) \times depth_{scale} + \Delta_k & \text{otherwise} \end{cases} \\a(x, y) &= \begin{cases} a(x, y) & \text{for array images} \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

where:

$$width_{scale} = width_{level}$$

$$height_{scale} = height_{level}$$

$$depth_{scale} = depth_{level}$$

for conventional images, and:

$$width_{scale} = width_{level} - 1$$

$$height_{scale} = height_{level} - 1$$

$$depth_{scale} = depth_{level} - 1$$

for corner-sampled images.

and where  $(\Delta_i, \Delta_j, \Delta_k)$  are taken from the image instruction if it includes a `ConstOffset` or `Offset` operand, otherwise they are taken to be zero.

Operations then proceed to Unnormalized Texel Coordinate Operations.

## 15.7. Unnormalized Texel Coordinate Operations

### 15.7.1. (u,v,w,a) to (i,j,k,l,n) Transformation And Array Layer Selection

The unnormalized texel coordinates are transformed to integer texel coordinates relative to the selected mipmap level.

The layer index  $l$  is computed as:

$$l = \text{clamp}(\text{RNE}(a), 0, \text{layerCount} - 1) + \text{baseArrayLayer}$$

where `layerCount` is the number of layers in the image subresource range of the image view, `baseArrayLayer` is the first layer from the subresource range, and where:

$$\text{RNE}(a) = \begin{cases} \text{roundTiesToEven}(a) & \text{preferred, from IEEE Std 754-2008 Floating-Point Arithmetic} \\ \lfloor a + 0.5 \rfloor & \text{alternative} \end{cases}$$

The sample index  $n$  is assigned the value zero.

Nearest filtering (`VK_FILTER_NEAREST`) computes the integer texel coordinates that the unnormalized coordinates lie within:

$$\begin{aligned} i &= \lfloor u + shift \rfloor \\ j &= \lfloor v + shift \rfloor \\ k &= \lfloor w + shift \rfloor \end{aligned}$$

where:

$$\text{shift} = 0.0$$

for conventional images, and:

$$\text{shift} = 0.5$$

for corner-sampled images.

Linear filtering (`VK_FILTER_LINEAR`) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of  $i_0$  or  $i_1$ ,  $j_0$  or  $j_1$ ,  $k_0$  or  $k_1$ , as well as weights  $\alpha$ ,  $\beta$ , and  $\gamma$ .

$$\begin{aligned} i_0 &= \lfloor u - shift \rfloor \\ i_1 &= i_0 + 1 \\ j_0 &= \lfloor v - shift \rfloor \\ j_1 &= j_0 + 1 \\ k_0 &= \lfloor w - shift \rfloor \\ k_1 &= k_0 + 1 \end{aligned}$$

$$\begin{aligned}\alpha &= \text{frac}(u - \text{shift}) \\ \beta &= \text{frac}(v - \text{shift}) \\ \gamma &= \text{frac}(w - \text{shift})\end{aligned}$$

where:

`shift = 0.5`

for conventional images, and:

`shift = 0.0`

for corner-sampled images, and where:

$$\text{frac}(x) = x - \lfloor x \rfloor$$

where the number of fraction bits retained is specified by `VkPhysicalDeviceLimits::subTexelPrecisionBits`.

Cubic filtering (`VK_FILTER_CUBIC_EXT`) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of  $i_0, i_1, i_2$  or  $i_3, j_0, j_1, j_2$  or  $j_3, k_0, k_1, k_2$  or  $k_3$ , as well as weights  $\alpha, \beta$ , and  $\gamma$ .

$$\begin{aligned}i_0 &= \lfloor u - \frac{3}{2} \rfloor & i_1 &= i_0 + 1 & i_2 &= i_1 + 1 & i_3 &= i_2 + 1 \\ j_0 &= \lfloor v - \frac{3}{2} \rfloor & j_1 &= j_0 + 1 & j_2 &= j_1 + 1 & j_3 &= j_2 + 1 \\ k_0 &= \lfloor w - \frac{3}{2} \rfloor & k_1 &= k_0 + 1 & k_2 &= k_1 + 1 & k_3 &= k_2 + 1\end{aligned}$$

$$\begin{aligned}\alpha &= \text{frac}\left(u - \frac{1}{2}\right) \\ \beta &= \text{frac}\left(v - \frac{1}{2}\right) \\ \gamma &= \text{frac}\left(w - \frac{1}{2}\right)\end{aligned}$$

where:

$$\text{frac}(x) = x - \lfloor x \rfloor$$

where the number of fraction bits retained is specified by `VkPhysicalDeviceLimits::subTexelPrecisionBits`.

## 15.8. Integer Texel Coordinate Operations

Integer texel coordinate operations **may** supply a LOD which texels are to be read from or written to using the optional SPIR-V operand `Lod`. If the `Lod` is provided then it **must** be an integer.

The image level selected is:

$$d = level_{base} + \begin{cases} Lod & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases}$$

If  $d$  does not lie in the range [`baseMipLevel`, `baseMipLevel` + `levelCount`) then any values fetched are undefined, and any writes are discarded.

## 15.9. Image Sample Operations

### 15.9.1. Wrapping Operation

[Cube](#) images ignore the wrap modes specified in the sampler. Instead, if `VK_FILTER_NEAREST` is used within a mip level then `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` is used, and if `VK_FILTER_LINEAR` is used within a mip level then sampling at the edges is performed as described earlier in the [Cube map edge handling](#) section.

The first integer texel coordinate  $i$  is transformed based on the `addressModeU` parameter of the sampler.

$$i = \begin{cases} i \bmod size & \text{for repeat} \\ (size - 1) - \text{mirror}((i \bmod (2 \times size)) - size) & \text{for mirrored repeat} \\ \text{clamp}(i, 0, size - 1) & \text{for clamp to edge} \\ \text{clamp}(i, -1, size) & \text{for clamp to border} \\ \text{clamp}(\text{mirror}(i), 0, size - 1) & \text{for mirror clamp to edge} \end{cases}$$

where:

$$\text{mirror}(n) = \begin{cases} n & \text{for } n \geq 0 \\ -(1 + n) & \text{otherwise} \end{cases}$$

$j$  (for 2D and Cube image) and  $k$  (for 3D image) are similarly transformed based on the `addressModeV` and `addressModeW` parameters of the sampler, respectively.

### 15.9.2. Texel Gathering

SPIR-V instructions with [Gather](#) in the name return a vector derived from 4 texels in the base level of the image view. The rules for the `VK_FILTER_LINEAR` minification filter are applied to identify the four selected texels. Each texel is then converted to an RGBA value according to [conversion to RGBA](#) and then [swizzled](#). A four-component vector is then assembled by taking the component indicated by the `Component` value in the instruction from the swizzled color value of the four texels. If the operation does not use the `ConstOffsets` image operand then the four texels form the  $2 \times 2$  rectangle used for texture filtering:

$$\begin{aligned} \tau[R] &= \tau_{i0j1}[level_{base}][comp] \\ \tau[G] &= \tau_{i1j1}[level_{base}][comp] \\ \tau[B] &= \tau_{i1j0}[level_{base}][comp] \\ \tau[A] &= \tau_{i0j0}[level_{base}][comp] \end{aligned}$$

If the operation does use the `ConstOffsets` image operand then the offsets allow a custom filter to be defined:

$$\begin{aligned}\tau[R] &= \tau_{i0j0} + \Delta_0[level_{base}][comp] \\ \tau[G] &= \tau_{i0j0} + \Delta_1[level_{base}][comp] \\ \tau[B] &= \tau_{i0j0} + \Delta_2[level_{base}][comp] \\ \tau[A] &= \tau_{i0j0} + \Delta_3[level_{base}][comp]\end{aligned}$$

where:

$$\tau[level_{base}][comp] = \begin{cases} \tau[level_{base}][R], & \text{for } comp = 0 \\ \tau[level_{base}][G], & \text{for } comp = 1 \\ \tau[level_{base}][B], & \text{for } comp = 2 \\ \tau[level_{base}][A], & \text{for } comp = 3 \end{cases}$$

*comp* from SPIR-V operand Component

**OpImage\*Gather** must not be used on a sampled image with **sampler Y'CbCr conversion** enabled.

### 15.9.3. Texel Filtering

Texel filtering is first performed for each level (either  $d$  or  $d_{hi}$  and  $d_{lo}$ ).

If  $\lambda$  is less than or equal to zero, the texture is said to be *magnified*, and the filter mode within a mip level is selected by the **magFilter** in the sampler. If  $\lambda$  is greater than zero, the texture is said to be *minified*, and the filter mode within a mip level is selected by the **minFilter** in the sampler.

#### Texel Nearest Filtering

Within a mip level, **VK\_FILTER\_NEAREST** filtering selects a single value using the  $(i, j, k)$  texel coordinates, with all texels taken from layer 1.

$$\tau[level] = \begin{cases} \tau_{ijk}[level], & \text{for 3D image} \\ \tau_{ij}[level], & \text{for 2D or Cube image} \\ \tau_i[level], & \text{for 1D image} \end{cases}$$

#### Texel Linear Filtering

Within a mip level, **VK\_FILTER\_LINEAR** filtering combines 8 (for 3D), 4 (for 2D or Cube), or 2 (for 1D) texel values, together with their linear weights. The linear weights are derived from the fractions computed earlier:

$$\begin{aligned}w_{i_0} &= (1 - \alpha) \\ w_{i_1} &= (\alpha) \\ w_{j_0} &= (1 - \beta) \\ w_{j_1} &= (\beta) \\ w_{k_0} &= (1 - \gamma) \\ w_{k_1} &= (\gamma)\end{aligned}$$

The values of multiple texels, together with their weights, are combined to produce a filtered value.

The **VkSamplerReductionModeCreateInfoEXT::reductionMode** can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the **reductionMode** is set (explicitly or implicitly) to

`VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT`, a weighted average is computed:

$$\begin{aligned}\tau_{3D} &= \sum_{k=k_0}^{k_1} \sum_{j=j_0}^{j_1} \sum_{i=i_0}^{i_1} (w_i)(w_j)(w_k)\tau_{ijk} \\ \tau_{2D} &= \sum_{j=j_0}^{j_1} \sum_{i=i_0}^{i_1} (w_i)(w_j)\tau_{ij} \\ \tau_{1D} &= \sum_{i=i_0}^{i_1} (w_i)\tau_i\end{aligned}$$

However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT`, the process operates on the above set of multiple texels, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the set of texels with non-zero weights.

## Texel Cubic Filtering

Within a mip level, `VK_FILTER_CUBIC_EXT`, filtering computes a weighted average of 64 (for 3D), 16 (for 2D), or 4 (for 1D) texel values, together with their Catmull-Rom weights.

Catmull-Rom weights are derived from the fractions computed earlier.

$$\begin{aligned}[w_{i_0} \ w_{i_1} \ w_{i_2} \ w_{i_3}] &= \frac{1}{2} [1 \ \alpha \ \alpha^2 \ \alpha^3] \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \\ [w_{j_0} \ w_{j_1} \ w_{j_2} \ w_{j_3}] &= \frac{1}{2} [1 \ \beta \ \beta^2 \ \beta^3] \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \\ [w_{k_0} \ w_{k_1} \ w_{k_2} \ w_{k_3}] &= \frac{1}{2} [1 \ \gamma \ \gamma^2 \ \gamma^3] \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix}\end{aligned}$$

The values of multiple texels, together with their weights, are combined to produce a filtered value.

The `VkSamplerReductionModeCreateInfoEXT::reductionMode` can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the `reductionMode` is set (explicitly or implicitly) to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT`, a weighted average is computed:

$$\begin{aligned}\tau_{3D} &= \sum_{k=k_0}^{k_3} \sum_{j=j_0}^{j_3} \sum_{i=i_0}^{i_3} (w_i)(w_j)(w_k)\tau_{ijk} \\ \tau_{2D} &= \sum_{j=j_0}^{j_3} \sum_{i=i_0}^{i_3} (w_i)(w_j)\tau_{ij} \\ \tau_{1D} &= \sum_{i=i_0}^{i_3} (w_i)\tau_i\end{aligned}$$

However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT`, the process operates on the above set of multiple texels, together with their weights, computing a component-wise minimum or maximum, respectively, of

the components of the set of texels with non-zero weights.

## Texel Mipmap Filtering

`VK_SAMPLER_MIPMAP_MODE_NEAREST` filtering returns the value of a single mipmap level,

$$\tau = \tau[d].$$

`VK_SAMPLER_MIPMAP_MODE_LINEAR` filtering combines the values of multiple mipmap levels ( $\tau[hi]$  and  $\tau[lo]$ ), together with their linear weights.

The linear weights are derived from the fraction computed earlier:

$$w_{hi} = (1 - \delta)$$
$$w_{lo} = (\delta)$$

The values of multiple mipmap levels, together with their weights, are combined to produce a final filtered value.

The `VkSamplerReductionModeCreateInfoEXT::reductionMode` can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the `reductionMode` is set (explicitly or implicitly) to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT`, a weighted average is computed:

$$\tau = (w_{hi})\tau[hi] + (w_{lo})\tau[lo]$$

## Texel Anisotropic Filtering

Anisotropic filtering is enabled by the `anisotropyEnable` in the sampler. When enabled, the image filtering scheme accounts for a degree of anisotropy.

The particular scheme for anisotropic texture filtering is implementation dependent. Implementations **should** consider the `magFilter`, `minFilter` and `mipmapMode` of the sampler to control the specifics of the anisotropic filtering scheme used. In addition, implementations **should** consider `minLod` and `maxLod` of the sampler.

The following describes one particular approach to implementing anisotropic filtering for the 2D Image case, implementations **may** choose other methods:

Given a `magFilter`, `minFilter` of `VK_FILTER_LINEAR` and a `mipmapMode` of `VK_SAMPLER_MIPMAP_MODE_NEAREST`:

Instead of a single isotropic sample, N isotropic samples are be sampled within the image footprint of the image level d to approximate an anisotropic filter. The sum  $\tau_{2D_{aniso}}$  is defined using the single isotropic  $\tau_{2D}(u,v)$  at level d.

$$\tau_{2D_{aniso}} = \frac{1}{N} \sum_{i=1}^N \tau_{2D}\left(u\left(x - \frac{1}{2} + \frac{i}{N+1}, y\right), \left(v\left(x - \frac{1}{2} + \frac{i}{N+1}\right), y\right)\right), \quad \text{when } \rho_x > \rho_y$$
$$\tau_{2D_{aniso}} = \frac{1}{N} \sum_{i=1}^N \tau_{2D}\left(u\left(x, y - \frac{1}{2} + \frac{i}{N+1}\right), \left(v\left(x, y - \frac{1}{2} + \frac{i}{N+1}\right)\right)\right), \quad \text{when } \rho_y \geq \rho_x$$

When `VkSamplerReductionModeCreateInfoEXT::reductionMode` is set to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT`, the above summation is used. However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT`, the process operates on the above values, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the values with non-zero weights.

## 15.10. Texel Footprint Evaluation

The SPIR-V instruction `OpImageSampleFootprintNV` evaluates the set of texels from a single mip level that would be accessed during a `texel filtering` operation. In addition to the inputs that would be accepted by an equivalent `OpImageSample*` instruction, `OpImageSampleFootprintNV` accepts two additional inputs. The `Granularity` input is an integer identifying the size of texel groups used to evaluate the footprint. Each bit in the returned footprint mask corresponds to an aligned block of texels whose size is given by the following table:

*Table 28. Texel footprint granularity values*

Granularity	Dim = 2D	Dim = 3D
0	unsupported	unsupported
1	2x2	2x2x2
2	4x2	unsupported
3	4x4	4x4x2
4	8x4	unsupported
5	8x8	unsupported
6	16x8	unsupported
7	16x16	unsupported
8	unsupported	unsupported
9	unsupported	unsupported
10	unsupported	16x16x16
11	64x64	32x16x16
12	128x64	32x32x16
13	128x128	32x32x32
14	256x128	64x32x32
15	256x256	unsupported

The `Coarse` input is used to select between the two mip levels that **may** be accessed during texel filtering when using a `mipmapMode` of `VK_SAMPLER_MIPMAP_MODE_LINEAR`. When filtering between two mip levels, a `Coarse` value of `true` requests the footprint in the lower-resolution mip level (higher level number), while `false` requests the footprint in the higher-resolution mip level. If texel filtering would access only a single mip level, the footprint in that level would be returned when `Coarse` is set to `false`; an empty footprint would be returned when `Coarse` is set to `true`.

The footprint for `OpImageSampleFootprintNV` is returned in a structure with six members:

- The first member is a boolean value that is true if the texel filtering operation would access only a single mip level.
- The second member is a two- or three-component integer vector holding the footprint anchor location. For two-dimensional images, the returned components are in units of eight texel groups. For three-dimensional images, the returned components are in units of four texel groups.
- The third member is a two- or three-component integer vector holding a footprint offset relative to the anchor. All returned components are in units of texel groups.
- The fourth member is a two-component integer vector mask, which holds a bitfield identifying the set of texel groups in an 8x8 or 4x4x4 neighborhood relative to the anchor and offset.
- The fifth member is an integer identifying the mip level containing the footprint identified by the anchor, offset, and mask.
- The sixth member is an integer identifying the granularity of the returned footprint.

For footprints in two-dimensional images ([Dim2D](#)), the mask returned by [OpImageSampleFootprintNV](#) indicates whether each texel group in a 8x8 local neighborhood of texel groups would have one or more texels accessed during texel filtering. In the mask, the texel group with local group coordinates ( $lgx, lgy$ ) is considered covered if and only if

$$0 \neq ((mask.x + (mask.y << 32)) \& (1 << (lgy \times 8 + lgx)))$$

where:

- $0 \leq lgx \leq 8$  and  $0 \leq lgy \leq 8$ ; and
- $mask$  is the returned two-component mask.

The local group with coordinates ( $lgx, lgy$ ) in the mask is considered covered if and only if the texel filtering operation would access one or more texels  $\tau_{ij}$  in the returned miplevel where:

$$\begin{aligned} i0 &= \begin{cases} gran.x \times (8 \times anchor.x + lgx), & \text{if } lgx + offset.x < 8 \\ gran.x \times (8 \times (anchor.x - 1) + lgx), & \text{otherwise} \end{cases} \\ i1 &= i0 + gran.x - 1 \\ j0 &= \begin{cases} gran.y \times (8 \times anchor.y + lgy), & \text{if } lgy + offset.y < 8 \\ gran.y \times (8 \times (anchor.y - 1) + lgy), & \text{otherwise} \end{cases} \\ j1 &= j0 + gran.y - 1 \end{aligned}$$

and

- $i0 \leq i \leq i1$  and  $j0 \leq j \leq j1$ ;
- $gran$  is a two-component vector holding the width and height of the texel group identified by the granularity;
- $anchor$  is the returned two-component anchor vector; and
- $offset$  is the returned two-component offset vector.

For footprints in three-dimensional images ([Dim3D](#)), the mask returned by [OpImageSampleFootprintNV](#) indicates whether each texel group in a 4x4x4 local neighborhood of texel groups would have one or more texels accessed during texel filtering. In the mask, the texel group with local group

coordinates  $(lgx, lgy, lgz)$ , is considered covered if and only if:

$$0 \neq ((mask.x + (mask.y \ll 32)) \& (1 \ll (lgz \times 16 + lgy \times 4 + lgx)))$$

where:

- $0 \leq lgx < 4$ ,  $0 \leq lgy < 4$ , and  $0 \leq lgz < 4$ ; and
- *mask* is the returned two-component mask.

The local group with coordinates  $(lgx, lgy, lgz)$  in the mask is considered covered if and only if the texel filtering operation would access one or more texels  $\tau_{ijk}$  in the returned mplevel where:

$$\begin{aligned} i0 &= \begin{cases} gran.x \times (4 \times anchor.x + lgx), & \text{if } lgx + offset.x < 4 \\ gran.x \times (4 \times (anchor.x - 1) + lgx), & \text{otherwise} \end{cases} \\ i1 &= i0 + gran.x - 1 \\ j0 &= \begin{cases} gran.y \times (4 \times anchor.y + lgy), & \text{if } lgy + offset.y < 4 \\ gran.y \times (4 \times (anchor.y - 1) + lgy), & \text{otherwise} \end{cases} \\ j1 &= j0 + gran.y - 1 \\ k0 &= \begin{cases} gran.z \times (4 \times anchor.z + lgz), & \text{if } lgz + offset.z < 4 \\ gran.z \times (4 \times (anchor.z - 1) + lgz), & \text{otherwise} \end{cases} \\ k1 &= k0 + gran.z - 1 \end{aligned}$$

and

- $i0 \leq i \leq i1$ ,  $j0 \leq j \leq j1$ ,  $k0 \leq k \leq k1$ ;
- *gran* is a three-component vector holding the width, height, and depth of the texel group identified by the granularity;
- *anchor* is the returned three-component anchor vector; and
- *offset* is the returned three-component offset vector.

If the sampler used by `OpImageSampleFootprintNV` enables anisotropic texel filtering via `anisotropyEnable`, it is possible that the set of texel groups accessed in a mip level may be too large to be expressed using an 8x8 or 4x4x4 mask using the granularity requested in the instruction. In this case, the implementation uses a texel group larger than the requested granularity. When a larger texel group size is used, `OpImageSampleFootprintNV` returns an integer granularity value that **can** be interpreted in the same manner as the granularity value provided to the instruction to determine the texel group size used. If anisotropic texel filtering is disabled in the sampler, or if an anisotropic footprint can be represented as an 8x8 or 4x4x4 mask with the requested granularity, `OpImageSampleFootprintNV` will use the requested granularity as-is and return a granularity value of zero.

`OpImageSampleFootprintNV` supports only two- and three-dimensional image accesses (`Dim2D` and `Dim3D`) and the footprint returned is undefined if a sampler uses an addressing mode other than `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

## 15.11. Image Operation Steps

Each step described in this chapter is performed by a subset of the image instructions:

- Texel Input Validation Operations, Format Conversion, Texel Replacement, Conversion to RGBA, and Component Swizzle: Performed by all instructions except `OpImageWrite`.
- Depth Comparison: Performed by `OpImage*Dref` instructions.
- All Texel output operations: Performed by `OpImageWrite`.
- Projection: Performed by all `OpImage*Proj` instructions.
- Derivative Image Operations, Cube Map Operations, Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection, and Texel Anisotropic Filtering: Performed by all `OpImageSample*` and `OpImageSparseSample*` instructions.
- (s,t,r,q,a) to (u,v,w,a) Transformation, Wrapping, and (u,v,w,a) to (i,j,k,l,n) Transformation And Array Layer Selection: Performed by all `OpImageSample`, `OpImageSparseSample`, and `OpImage*Gather` instructions.
- Texel Gathering: Performed by `OpImage*Gather` instructions.
- Texel Footprint Evaluation: Performed by `OpImageSampleFootprint` instructions.
- Texel Filtering: Performed by all `OpImageSample*` and `OpImageSparseSample*` instructions.
- Sparse Residency: Performed by all `OpImageSparse*` instructions.

# Chapter 16. Fragment Density Map Operations

## 16.1. Fragment Density Map Operations Overview

When a fragment is generated in a render pass that has a fragment density map attachment, its area is determined by the properties of the local framebuffer region that the fragment occupies. The framebuffer is divided into a uniform grid of these local regions, and their fragment area property is derived from the density map with the following operations:

- Fetch density value
  - Component swizzle
  - Component mapping
- Fragment area conversion
  - Fragment area filter
  - Fragment area clamp

## 16.2. Fetch Density Value

Each local framebuffer region at center coordinate (x,y) fetches a texel from the fragment density map at integer coordinates:

$$i = \lfloor \frac{x}{fragmentDensityTexelSize_{width}} \rfloor$$

$$j = \lfloor \frac{y}{fragmentDensityTexelSize_{height}} \rfloor$$

Where the size of each region in the framebuffer is:

$$fragmentDensityTexelSize_{width} = 2^{\lceil \log_2(\frac{framebuffer_{width}}{fragmentDensityMap_{width}}) \rceil}$$

$$fragmentDensityTexelSize_{height} = 2^{\lceil \log_2(\frac{framebuffer_{height}}{fragmentDensityMap_{height}}) \rceil}$$

This region is subject to the limits in `VkPhysicalDeviceFragmentDensityMapPropertiesEXT` and therefore the final region size is clamped:

`fragmentDensityTexelSize_{width} = clamp(fragmentDensityTexelSize_{width}, minFragmentDensityTexelSize_{width}, maxFragmentDensityTexelSize_{width})`

`fragmentDensityTexelSize_{height} = clamp(fragmentDensityTexelSize_{height}, minFragmentDensityTexelSize_{height}, maxFragmentDensityTexelSize_{height})`

When multiview is enabled for the render pass and the fragment density map attachment view was created with `layerCount` greater than 1, the density map layer that the texel is fetched from is:

`layer = baseArrayLayer + ViewIndex`

Otherwise:

*layer = baseArrayLayer*

The texel fetched from the density map at (i,j,layer) is next converted to density with the following operations.

### 16.2.1. Component Swizzle

The `components` member of [VkImageViewCreateInfo](#) is applied to the fetched texel as defined in [Image component swizzle](#).

### 16.2.2. Component Mapping

The swizzled texel's components are mapped to a density value:

$$densityValue_{xy} = (C'r, C'g)$$

## 16.3. Fragment Area Conversion

Fragment area for the framebuffer region is undefined if the density fetched is not a normalized floating-point value greater than `0.0`. Otherwise, the fetched fragment area for that region is derived as:

$$fragmentArea_{wh} = \frac{1.0}{densityValue_{xy}}$$

### 16.3.1. Fragment Area Filter

Optionally, the implementation **may** fetch additional density map texels in an implementation defined window around (i,j). The texels follow the standard conversion steps up to and including [fragment area conversion](#).

A single fetched fragment area for the framebuffer region is chosen by the implementation and **must** have an area between the *min* and *max* areas of the fetched set.

### 16.3.2. Fragment Area Clamp

The implementation **may** clamp the fetched fragment area to one that it supports. The clamped fragment area **must** have a size less than or equal to the original fetched value. Implementations **may** vary the supported set of fragment areas per framebuffer region. Fragment area (1,1) **must** always be in the supported set.

#### Note



For example, if the fetched fragment area is (1,4) but the implementation only supports areas of {(1,1),(2,2)}, it could choose to clamp the area to (2,2) since it has the same size as (1,4). While this would produce fragments that have lower quality strictly in the x-axis, the overall density is maintained.

The clamped fragment area is assigned to the corresponding framebuffer region.

# Chapter 17. Queries

*Queries* provide a mechanism to return information about the processing of a sequence of Vulkan commands. Query operations are asynchronous, and as such, their results are not returned immediately. Instead, their results, and their availability status are stored in a [Query Pool](#). The state of these queries **can** be read back on the host, or copied to a buffer object on the device.

The supported query types are [Occlusion Queries](#), [Pipeline Statistics Queries](#), and [Timestamp Queries](#). Intel performance queries are also supported if the associated extension is available.

## 17.1. Query Pools

Queries are managed using *query pool* objects. Each query pool is a collection of a specific number of queries of a particular type.

Query pools are represented by `VkQueryPool` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkQueryPool)
```

To create a query pool, call:

```
VkResult vkCreateQueryPool(  
    VkDevice device,  
    const VkQueryPoolCreateInfo* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkQueryPool* pQueryPool);
```

- `device` is the logical device that creates the query pool.
- `pCreateInfo` is a pointer to a `VkQueryPoolCreateInfo` structure containing the number and type of queries to be managed by the pool.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pQueryPool` is a pointer to a `VkQueryPool` handle in which the resulting query pool object is returned.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkQueryPoolCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pQueryPool` **must** be a valid pointer to a `VkQueryPool` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkQueryPoolCreateInfo` structure is defined as:

```
typedef struct VkQueryPoolCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkQueryPoolCreateFlags    flags;
    VkQueryType               queryType;
    uint32_t                  queryCount;
    VkQueryPipelineStatisticFlags pipelineStatistics;
} VkQueryPoolCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `queryType` is a `VkQueryType` value specifying the type of queries managed by the pool.
- `queryCount` is the number of queries managed by the pool.
- `pipelineStatistics` is a bitmask of `VkQueryPipelineStatisticFlagBits` specifying which counters will be returned in queries on the new pool, as described below in [Pipeline Statistics Queries](#).

`pipelineStatistics` is ignored if `queryType` is not `VK_QUERY_TYPE_PIPELINE_STATISTICS`.

## Valid Usage

- If the `pipeline statistics queries` feature is not enabled, `queryType` **must** not be `VK_QUERY_TYPE_PIPELINE_STATISTICS`
- If `queryType` is `VK_QUERY_TYPE_PIPELINE_STATISTICS`, `pipelineStatistics` **must** be a valid combination of `VkQueryPipelineStatisticFlagBits` values
- If `queryType` is `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `pNext` chain **must** contain a structure of type `VkQueryPoolPerformanceCreateInfoKHR`
- `queryCount` **must** be greater than 0

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkQueryPoolPerformanceCreateInfoKHR`
- `flags` **must** be `0`
- `queryType` **must** be a valid `VkQueryType` value

```
typedef VkFlags VkQueryPoolCreateFlags;
```

`VkQueryPoolCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The `VkQueryPoolPerformanceCreateInfoKHR` structure is defined as:

```
typedef struct VkQueryPoolPerformanceCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t queueFamilyIndex;
    uint32_t counterIndexCount;
    const uint32_t* pCounterIndices;
} VkQueryPoolPerformanceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `queueFamilyIndex` is the queue family index to create this performance query pool for.
- `counterIndexCount` is size of the `pCounterIndices` array.
- `pCounterIndices` is the array of indices into the `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR::pCounters` to enable in this performance query pool.

## Valid Usage

- `queueFamilyIndex` **must** be a valid queue family index of the device.
- The `performanceCounterQueryPools` feature **must** be enabled
- Each element of `pCounterIndices` **must** be in the range of counters reported by `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR` for the queue family specified in `queueFamilyIndex`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR`
- `pCounterIndices` **must** be a valid pointer to an array of `counterIndexCount uint32_t` values
- `counterIndexCount` **must** be greater than `0`

To query the number of passes required to query a performance query pool on a physical device, call:

```
void vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR(  
    VkPhysicalDevice           physicalDevice,  
    const VkQueryPoolPerformanceCreateInfoKHR* pPerformanceQueryCreateInfo,  
    uint32_t*                  pNumPasses);
```

- `physicalDevice` is the handle to the physical device whose queue family performance query counter properties will be queried
- `pPerformanceQueryCreateInfo` is a pointer to an `VkQueryPoolPerformanceCreateInfoKHR` of the performance query that is to be created
- `pNumPasses` is a pointer to an integer related to the number of passes required to query the performance query pool, as described below

The `pPerformanceQueryCreateInfo` member `VkQueryPoolPerformanceCreateInfoKHR::queueFamilyIndex` **must** be a queue family of `physicalDevice`. The number of passes required to capture the counters specified in the `pPerformanceQueryCreateInfo` member `VkQueryPoolPerformanceCreateInfoKHR::pCounters` is returned in `pNumPasses`.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pPerformanceQueryCreateInfo` **must** be a valid pointer to a `VkQueryPoolPerformanceCreateInfoKHR` structure
- `pNumPasses` **must** be a valid pointer to a `uint32_t` value

To destroy a query pool, call:

```
void vkDestroyQueryPool(  
    VkDevice           device,  
    VkQueryPool        queryPool,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the query pool.
- `queryPool` is the query pool to destroy.

- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- All submitted commands that refer to `queryPool` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `queryPool` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `queryPool` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `queryPool` is not `VK_NULL_HANDLE`, `queryPool` **must** be a valid `VkQueryPool` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `queryPool` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `queryPool` **must** be externally synchronized

Possible values of `VkQueryPoolCreateInfo::queryType`, specifying the type of queries managed by the pool, are:

```
typedef enum VkQueryType {
    VK_QUERY_TYPE_OCCLUSION = 0,
    VK_QUERY_TYPE_PIPELINE_STATISTICS = 1,
    VK_QUERY_TYPE_TIMESTAMP = 2,
    VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT = 1000028004,
    VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR = 1000116000,
    VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV = 1000165000,
    VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL = 1000210000,
    VK_QUERY_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkQueryType;
```

- `VK_QUERY_TYPE_OCCLUSION` specifies an [occlusion query](#).
- `VK_QUERY_TYPE_PIPELINE_STATISTICS` specifies a [pipeline statistics query](#).
- `VK_QUERY_TYPE_TIMESTAMP` specifies a [timestamp query](#).
- `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` specifies a [performance query](#).

- `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` specifies a transform feedback query.
- `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV` specifies a ray tracing acceleration structure size query.
- `VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL` specifies a Intel performance query.

## 17.2. Query Operation

The operation of queries is controlled by the commands `vkCmdBeginQuery`, `vkCmdEndQuery`, `vkCmdBeginQueryIndexedEXT`, `vkCmdEndQueryIndexedEXT`, `vkCmdResetQueryPool`, `vkCmdCopyQueryPoolResults`, and `vkCmdWriteTimestamp`.

In order for a `VkCommandBuffer` to record query management commands, the queue family for which its `VkCommandPool` was created **must** support the appropriate type of operations (graphics, compute) suitable for the query type of a given query pool.

Each query in a query pool has a status that is either *unavailable* or *available*, and also has state to store the numerical results of a query operation of the type requested when the query pool was created. Resetting a query via `vkCmdResetQueryPool` or `vkResetQueryPoolEXT` sets the status to unavailable and makes the numerical results undefined. Performing a query operation with `vkCmdBeginQuery` and `vkCmdEndQuery` changes the status to available when the query *finishes*, and updates the numerical results. Both the availability status and numerical results are retrieved by calling either `vkGetQueryPoolResults` or `vkCmdCopyQueryPoolResults`.

Query commands, for the same query and submitted to the same queue, execute in their entirety in **submission order**, relative to each other. In effect there is an implicit execution dependency from each such query command to all query command previously submitted to the same queue. There is one significant exception to this; if the `flags` parameter of `vkCmdCopyQueryPoolResults` does not include `VK_QUERY_RESULT_WAIT_BIT`, execution of `vkCmdCopyQueryPoolResults` **may** happen-before the results of `vkCmdEndQuery` are available.

After query pool creation, each query **must** be reset before it is used. Queries **must** also be reset between uses.

If a logical device includes multiple physical devices, then each command that writes a query **must** execute on a single physical device, and any call to `vkCmdBeginQuery` **must** execute the corresponding `vkCmdEndQuery` command on the same physical device.

To reset a range of queries in a query pool on a queue, call:

```
void vkCmdResetQueryPool(
    VkCommandBuffer
    VkQueryPool
    uint32_t
    uint32_t
                                commandBuffer,
                                queryPool,
                                firstQuery,
                                queryCount);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the handle of the query pool managing the queries being reset.

- `firstQuery` is the initial query index to reset.
- `queryCount` is the number of queries to reset.

When executed on a queue, this command sets the status of query indices [`firstQuery`, `firstQuery + queryCount - 1`] to unavailable.

## Valid Usage

- `firstQuery` **must** be less than the number of queries in `queryPool`
- The sum of `firstQuery` and `queryCount` **must** be less than or equal to the number of queries in `queryPool`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- Both of `commandBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics Compute	

To reset a range of queries in a query pool on the host, call:

```
void vkResetQueryPoolEXT(
    VkDevice device,
    VkQueryPool queryPool,
    uint32_t firstQuery,
    uint32_t queryCount);
```

- `device` is the logical device that owns the query pool.
- `queryPool` is the handle of the query pool managing the queries being reset.
- `firstQuery` is the initial query index to reset.
- `queryCount` is the number of queries to reset.

This command sets the status of query indices [`firstQuery`, `firstQuery + queryCount - 1`] to unavailable.

## Valid Usage

- The `hostQueryReset` feature **must** be enabled
- `firstQuery` **must** be less than the number of queries in `queryPool`
- The sum of `firstQuery` and `queryCount` **must** be less than or equal to the number of queries in `queryPool`
- Submitted commands that refer to the range specified by `firstQuery` and `queryCount` in `queryPool` **must** have completed execution
- The range of queries specified by `firstQuery` and `queryCount` in `queryPool` **must** not be in use by calls to `vkGetQueryPoolResults` or `vkResetQueryPoolEXT` in other threads

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `queryPool` **must** have been created, allocated, or retrieved from `device`

Once queries are reset and ready for use, query commands **can** be issued to a command buffer. Occlusion queries and pipeline statistics queries count events - drawn samples and pipeline stage invocations, respectively - resulting from commands that are recorded between a `vkCmdBeginQuery` command and a `vkCmdEndQuery` command within a specified command buffer, effectively scoping a set of drawing and/or dispatch commands. Timestamp queries write timestamps to a query pool. Performance queries record performance counters to a query pool.

A query **must** begin and end in the same command buffer, although if it is a primary command buffer, and the `inherited queries` feature is enabled, it **can** execute secondary command buffers during the query operation. For a secondary command buffer to be executed while a query is active, it **must** set the `occlusionQueryEnable`, `queryFlags`, and/or `pipelineStatistics` members of

[VkCommandBufferInheritanceInfo](#) to conservative values, as described in the [Command Buffer Recording](#) section. A query **must** either begin and end inside the same subpass of a render pass instance, or **must** both begin and end outside of a render pass instance (i.e. contain entire render pass instances).

If queries are used while executing a render pass instance that has multiview enabled, the query uses N consecutive query indices in the query pool (starting at [query](#)) where N is the number of bits set in the view mask in the subpass the query is used in. How the numerical results of the query are distributed among the queries is implementation-dependent. For example, some implementations **may** write each view's results to a distinct query, while other implementations **may** write the total result to the first query and write zero to the other queries. However, the sum of the results in all the queries **must** accurately reflect the total result of the query summed over all views. Applications **can** sum the results from all the queries to compute the total result.

Queries used with multiview rendering **must** not span subpasses, i.e. they **must** begin and end in the same subpass.

To begin a query, call:

```
void vkCmdBeginQuery(  
    VkCommandBuffer           commandBuffer,  
    VkQueryPool               queryPool,  
    uint32_t                  query,  
    VkQueryControlFlags       flags);
```

- [commandBuffer](#) is the command buffer into which this command will be recorded.
- [queryPool](#) is the query pool that will manage the results of the query.
- [query](#) is the query index within the query pool that will contain the results.
- [flags](#) is a bitmask of [VkQueryControlFlagBits](#) specifying constraints on the types of queries that **can** be performed.

If the [queryType](#) of the pool is [VK\\_QUERY\\_TYPE\\_OCCLUSION](#) and [flags](#) contains [VK\\_QUERY\\_CONTROL\\_PRECISE\\_BIT](#), an implementation **must** return a result that matches the actual number of samples passed. This is described in more detail in [Occlusion Queries](#).

Calling [vkCmdBeginQuery](#) is equivalent to calling [vkCmdBeginQueryIndexedEXT](#) with the [index](#) parameter set to zero.

After beginning a query, that query is considered *active* within the command buffer it was called in until that same query is ended. Queries active in a primary command buffer when secondary command buffers are executed are considered active for those secondary command buffers.

## Valid Usage

- `queryPool` **must** have been created with a `queryType` that differs from that of any queries that are `active` within `commandBuffer`
- All queries used by the command **must** be unavailable
- The `queryType` used to create `queryPool` **must** not be `VK_QUERY_TYPE_TIMESTAMP`
- If the `precise occlusion queries` feature is not enabled, or the `queryType` used to create `queryPool` was not `VK_QUERY_TYPE_OCCLUSION`, `flags` **must** not contain `VK_QUERY_CONTROL_PRECISE_BIT`
- `query` **must** be less than the number of queries in `queryPool`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_OCCLUSION`, the `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PIPELINE_STATISTICS` and any of the `pipelineStatistics` indicate graphics operations, the `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PIPELINE_STATISTICS` and any of the `pipelineStatistics` indicate compute operations, the `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- `commandBuffer` **must** not be a protected command buffer
- If called within a render pass instance, the sum of `query` and the number of bits set in the current subpass's view mask **must** be less than or equal to the number of queries in `queryPool`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` then `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackQueries` **must** be supported
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `profiling lock` **must** have been held before `vkBeginCommandBuffer` was called on `commandBuffer`.
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one of the counters used to create `queryPool` was `VK_QUERY_SCOPE_COMMAND_BUFFER_KHR`, the query begin **must** be the first recorded command in `commandBuffer`.
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one of the counters used to create `queryPool` was `VK_QUERY_SCOPE_RENDER_PASS_KHR`, the begin command **must** not be recorded within a render pass instance.
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and another query pool with a `queryType` `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` has been used within `commandBuffer`, its parent primary command buffer or secondary command buffer recorded within the same parent primary command buffer as `commandBuffer`, the `performanceCounterMultipleQueryPools` feature **must** be enabled.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `flags` **must** be a valid combination of `VkQueryControlFlagBits` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- Both of `commandBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

To begin an indexed query, call:

```
void vkCmdBeginQueryIndexedEXT(  
    VkCommandBuffer  
    VkQueryPool  
    uint32_t  
    VkQueryControlFlags  
    uint32_t  
        commandBuffer,  
        queryPool,  
        query,  
        flags,  
        index);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the query pool that will manage the results of the query.
- `query` is the query index within the query pool that will contain the results.
- `flags` is a bitmask of `VkQueryControlFlagBits` specifying constraints on the types of queries that can be performed.
- `index` is the query type specific index. When the query type is `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the index represents the vertex stream.

The `vkCmdBeginQueryIndexedEXT` command operates the same as the `vkCmdBeginQuery` command, except that it also accepts a query type specific `index` parameter.

## Valid Usage

- `queryPool` **must** have been created with a `queryType` that differs from that of any queries that are `active` within `commandBuffer`
- All queries used by the command **must** be unavailable
- The `queryType` used to create `queryPool` **must** not be `VK_QUERY_TYPE_TIMESTAMP`
- If the `precise occlusion queries` feature is not enabled, or the `queryType` used to create `queryPool` was not `VK_QUERY_TYPE_OCCLUSION`, `flags` **must** not contain `VK_QUERY_CONTROL_PRECISE_BIT`
- `query` **must** be less than the number of queries in `queryPool`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_OCCLUSION`, the `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PIPELINE_STATISTICS` and any of the `pipelineStatistics` indicate graphics operations, the `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PIPELINE_STATISTICS` and any of the `pipelineStatistics` indicate compute operations, the `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- `commandBuffer` **must** not be a protected command buffer
- If called within a render pass instance, the sum of `query` and the number of bits set in the current subpass's view mask **must** be less than or equal to the number of queries in `queryPool`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `index` parameter **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `index` **must** be zero
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` then `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackQueries` **must** be supported
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `profiling lock` **must** have been held before `vkBeginCommandBuffer` was called on `commandBuffer`.
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one of the counters used to create `queryPool` was `VK_QUERY_SCOPE_COMMAND_BUFFER_KHR`, the query begin **must** be the first recorded command in `commandBuffer`.
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one of the counters used to create `queryPool` was `VK_QUERY_SCOPE_RENDER_PASS_KHR`, the begin

command **must** not be recorded within a render pass instance.

- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and another query pool with a `queryType` `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` has been used within `commandBuffer`, its parent primary command buffer or secondary command buffer recorded within the same parent primary command buffer as `commandBuffer`, the `performanceCounterMultipleQueryPools` feature **must** be enabled.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `flags` **must** be a valid combination of `VkQueryControlFlagBits` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- Both of `commandBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

Bits which **can** be set in `vkCmdBeginQuery::flags`, specifying constraints on the types of queries that **can** be performed, are:

```
typedef enum VkQueryControlFlagBits {
    VK_QUERY_CONTROL_PRECISE_BIT = 0x00000001,
    VK_QUERY_CONTROL_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkQueryControlFlagBits;
```

- `VK_QUERY_CONTROL_PRECISE_BIT` specifies the precision of `occlusion` queries.

```
typedef VkFlags VkQueryControlFlags;
```

`VkQueryControlFlags` is a bitmask type for setting a mask of zero or more `VkQueryControlFlagBits`.

To end a query after the set of desired draw or dispatch commands is executed, call:

```
void vkCmdEndQuery(  
    VkCommandBuffer  
    VkQueryPool  
    uint32_t  
                                commandBuffer,  
                                queryPool,  
                                query);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the query pool that is managing the results of the query.
- `query` is the query index within the query pool where the result is stored.

Calling `vkCmdEndQuery` is equivalent to calling `vkCmdEndQueryIndexedEXT` with the `index` parameter set to zero.

As queries operate asynchronously, ending a query does not immediately set the query's status to available. A query is considered *finished* when the final results of the query are ready to be retrieved by `vkGetQueryPoolResults` and `vkCmdCopyQueryPoolResults`, and this is when the query's status is set to available.

Once a query is ended the query **must** finish in finite time, unless the state of the query is changed using other commands, e.g. by issuing a reset of the query.

## Valid Usage

- All queries used by the command **must** be `active`
- `query` **must** be less than the number of queries in `queryPool`
- `commandBuffer` **must** not be a protected command buffer
- If `vkCmdEndQuery` is called within a render pass instance, the sum of `query` and the number of bits set in the current subpass's view mask **must** be less than or equal to the number of queries in `queryPool`
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one or more of the counters used to create `queryPool` was `VK_QUERY_SCOPE_COMMAND_BUFFER_KHR`, the `vkCmdEndQuery` **must** be the last recorded command in `commandBuffer`.
- If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one or more of the counters used to create `queryPool` was `VK_QUERY_SCOPE_RENDER_PASS_KHR`, the `vkCmdEndQuery` **must** not be recorded within a render pass instance.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- Both of `commandBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

To end an indexed query after the set of desired draw or dispatch commands is recorded, call:

```
void vkCmdEndQueryIndexedEXT(  
    VkCommandBuffer  
    VkQueryPool  
    uint32_t  
    uint32_t  
                                commandBuffer,  
                                queryPool,  
                                query,  
                                index);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the query pool that is managing the results of the query.
- `query` is the query index within the query pool where the result is stored.
- `index` is the query type specific index.

The `vkCmdEndQueryIndexedEXT` command operates the same as the `vkCmdEndQuery` command, except that it also accepts a query type specific `index` parameter.

## Valid Usage

- All queries used by the command **must** be `active`
- `query` **must** be less than the number of queries in `queryPool`
- `commandBuffer` **must** not be a protected command buffer
- If `vkCmdEndQueryIndexedEXT` is called within a render pass instance, the sum of `query` and the number of bits set in the current subpass's view mask **must** be less than or equal to the number of queries in `queryPool`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `index` parameter **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `index` **must** be zero
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` `index` **must** equal the `index` used to begin the query

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- Both of `commandBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

An application **can** retrieve results either by requesting they be written into application-provided memory, or by requesting they be copied into a [VkBuffer](#). In either case, the layout in memory is defined as follows:

- The first query's result is written starting at the first byte requested by the command, and each subsequent query's result begins [stride](#) bytes later.
- Occlusion queries, pipeline statistics queries, transform feedback queries, and timestamp queries store results in a tightly packed array of unsigned integers, either 32- or 64-bits as requested by the command, storing the numerical results and, if requested, the availability status.
- Performance queries store results in a tightly packed array whose type is determined by the [unit](#) member of the corresponding [VkPerformanceCounterKHR](#).
- If [VK\\_QUERY\\_RESULT\\_WITH\\_AVAILABILITY\\_BIT](#) is used, the final element of each query's result is an integer indicating whether the query's result is available, with any non-zero value indicating that it is available.
- Occlusion queries write one integer value - the number of samples passed. Pipeline statistics queries write one integer value for each bit that is enabled in the [pipelineStatistics](#) when the pool is created, and the statistics values are written in bit order starting from the least significant bit. Timestamp queries write one integer value. Performance queries write one [VkPerformanceCounterResultKHR](#) value for each [VkPerformanceCounterKHR](#) in the query. Transform feedback queries write two integers; the first integer is the number of primitives successfully written to the corresponding transform feedback buffer and the second is the number of primitives output to the vertex stream, regardless of whether they were successfully captured or not. In other words, if the transform feedback buffer was sized too small for the number of primitives output by the vertex stream, the first integer represents the number of primitives actually written and the second is the number that would have been written if all the transform feedback buffers associated with that vertex stream were large enough.
- If more than one query is retrieved and [stride](#) is not at least as large as the size of the array of values corresponding to a single query, the values written to memory are undefined.

To retrieve status and results for a set of queries, call:

```
VkResult vkGetQueryPoolResults(  
    VkDevice  
    VkQueryPool  
    uint32_t  
    uint32_t  
    size_t  
    void*  
    VkDeviceSize  
    VkQueryResultFlags  
        device,  
        queryPool,  
        firstQuery,  
        queryCount,  
        dataSize,  
        pData,  
        stride,  
        flags);
```

- [device](#) is the logical device that owns the query pool.
- [queryPool](#) is the query pool managing the queries containing the desired results.
- [firstQuery](#) is the initial query index.

- `queryCount` is the number of queries to read.
- `dataSize` is the size in bytes of the buffer pointed to by `pData`.
- `pData` is a pointer to a user-allocated buffer where the results will be written
- `stride` is the stride in bytes between results for individual queries within `pData`.
- `flags` is a bitmask of `VkQueryResultFlagBits` specifying how and when results are returned.

The range of queries read is defined by [`firstQuery`, `firstQuery` + `queryCount` - 1]. For pipeline statistics queries, each query index in the pool contains one integer value for each bit that is enabled in `VkQueryPoolCreateInfo::pname` `pipelineStatistics` when the pool is created.

If no bits are set in `flags`, and all requested queries are in the available state, results are written as an array of 32-bit unsigned integer values. The behavior when not all queries are available, is described [below](#).

If `VK_QUERY_RESULT_64_BIT` is not set and the result overflows a 32-bit value, the value **may** either wrap or saturate. Similarly, if `VK_QUERY_RESULT_64_BIT` is set and the result overflows a 64-bit value, the value **may** either wrap or saturate.

If `VK_QUERY_RESULT_WAIT_BIT` is set, Vulkan will wait for each query to be in the available state before retrieving the numerical results for that query. In this case, `vkGetQueryPoolResults` is guaranteed to succeed and return `VK_SUCCESS` if the queries become available in a finite time (i.e. if they have been issued and not reset). If queries will never finish (e.g. due to being reset but not issued), then `vkGetQueryPoolResults` **may** not return in finite time.

If `VK_QUERY_RESULT_WAIT_BIT` and `VK_QUERY_RESULT_PARTIAL_BIT` are both not set then no result values are written to `pData` for queries that are in the unavailable state at the time of the call, and `vkGetQueryPoolResults` returns `VK_NOT_READY`. However, availability state is still written to `pData` for those queries if `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` is set.

#### Note

Applications **must** take care to ensure that use of the `VK_QUERY_RESULT_WAIT_BIT` bit has the desired effect.



For example, if a query has been used previously and a command buffer records the commands `vkCmdResetQueryPool`, `vkCmdBeginQuery`, and `vkCmdEndQuery` for that query, then the query will remain in the available state until `vkResetQueryPoolEXT` is called or the `vkCmdResetQueryPool` command executes on a queue. Applications **can** use fences or events to ensure that a query has already been reset before checking for its results or availability status. Otherwise, a stale value could be returned from a previous use of the query.

The above also applies when `VK_QUERY_RESULT_WAIT_BIT` is used in combination with `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT`. In this case, the returned availability status **may** reflect the result of a previous use of the query unless `vkResetQueryPoolEXT` is called or the `vkCmdResetQueryPool` command has been executed since the last use of the query.

*Note*



Applications **can** double-buffer query pool usage, with a pool per frame, and reset queries at the end of the frame in which they are read.

If `VK_QUERY_RESULT_PARTIAL_BIT` is set, `VK_QUERY_RESULT_WAIT_BIT` is not set, and the query's status is unavailable, an intermediate result value between zero and the final result value is written to `pData` for that query.

If `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` is set, the final integer value written for each query is non-zero if the query's status was available or zero if the status was unavailable. When `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` is used, implementations **must** guarantee that if they return a non-zero availability value then the numerical results **must** be valid, assuming the results are not reset by a subsequent command.

*Note*



Satisfying this guarantee **may** require careful ordering by the application, e.g. to read the availability status before reading the results.

## Valid Usage

- `firstQuery` **must** be less than the number of queries in `queryPool`
- If `VK_QUERY_RESULT_64_BIT` is not set in `flags` and the `queryType` used to create `queryPool` was not `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` then `pData` and `stride` **must** be multiples of 4
- If `VK_QUERY_RESULT_64_BIT` is set in `flags` then `pData` and `stride` **must** be multiples of 8
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, then `pData` and `stride` **must** be multiples of the size of `VkPerformanceCounterResultKHR`
- The sum of `firstQuery` and `queryCount` **must** be less than or equal to the number of queries in `queryPool`
- `dataSize` **must** be large enough to contain the result of each query, as described [here](#)
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TIMESTAMP`, `flags` **must** not contain `VK_QUERY_RESULT_PARTIAL_BIT`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, `flags` **must** not contain `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT`, `VK_QUERY_RESULT_PARTIAL_BIT` or `VK_QUERY_RESULT_64_BIT`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `queryPool` **must** have been recorded once for each pass as retrieved via a call to `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `pData` **must** be a valid pointer to an array of `dataSize` bytes
- `flags` **must** be a valid combination of `VkQueryResultFlagBits` values
- `dataSize` **must** be greater than `0`
- `queryPool` **must** have been created, allocated, or retrieved from `device`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_NOT_READY`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

Bits which **can** be set in `vkGetQueryPoolResults::flags` and `vkCmdCopyQueryPoolResults::flags`, specifying how and when results are returned, are:

```
typedef enum VkQueryResultFlagBits {
    VK_QUERY_RESULT_64_BIT = 0x00000001,
    VK_QUERY_RESULT_WAIT_BIT = 0x00000002,
    VK_QUERY_RESULT_WITH_AVAILABILITY_BIT = 0x00000004,
    VK_QUERY_RESULT_PARTIAL_BIT = 0x00000008,
    VK_QUERY_RESULT_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkQueryResultFlagBits;
```

- `VK_QUERY_RESULT_64_BIT` specifies the results will be written as an array of 64-bit unsigned integer values. If this bit is not set, the results will be written as an array of 32-bit unsigned integer values.
- `VK_QUERY_RESULT_WAIT_BIT` specifies that Vulkan will wait for each query's status to become available before retrieving its results.
- `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` specifies that the availability status accompanies the results.
- `VK_QUERY_RESULT_PARTIAL_BIT` specifies that returning partial results is acceptable.

```
typedef VkFlags VkQueryResultFlags;
```

`VkQueryResultFlags` is a bitmask type for setting a mask of zero or more `VkQueryResultFlagBits`.

To copy query statuses and numerical results directly to buffer memory, call:

```
void vkCmdCopyQueryPoolResults(  
    VkCommandBuffer  
    commandBuffer,  
    VkQueryPool  
    queryPool,  
    uint32_t  
    firstQuery,  
    uint32_t  
    queryCount,  
    VkBuffer  
    dstBuffer,  
    VkDeviceSize  
    dstOffset,  
    VkDeviceSize  
    stride,  
    VkQueryResultFlags  
    flags);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the query pool managing the queries containing the desired results.
- `firstQuery` is the initial query index.
- `queryCount` is the number of queries. `firstQuery` and `queryCount` together define a range of queries.
- `dstBuffer` is a `VkBuffer` object that will receive the results of the copy command.
- `dstOffset` is an offset into `dstBuffer`.
- `stride` is the stride in bytes between results for individual queries within `dstBuffer`. The required size of the backing memory for `dstBuffer` is determined as described above for `vkGetQueryPoolResults`.
- `flags` is a bitmask of `VkQueryResultFlagBits` specifying how and when results are returned.

`vkCmdCopyQueryPoolResults` is guaranteed to see the effect of previous uses of `vkCmdResetQueryPool` in the same queue, without any additional synchronization. Thus, the results will always reflect the most recent use of the query.

`flags` has the same possible values described above for the `flags` parameter of `vkGetQueryPoolResults`, but the different style of execution causes some subtle behavioral differences. Because `vkCmdCopyQueryPoolResults` executes in order with respect to other query commands, there is less ambiguity about which use of a query is being requested.

Results for all requested occlusion queries, pipeline statistics queries, transform feedback queries, and timestamp queries are written as 64-bit unsigned integer values if `VK_QUERY_RESULT_64_BIT` is set or 32-bit unsigned integer values otherwise. Performance queries store results in a tightly packed array whose type is determined by the `unit` member of the corresponding `VkPerformanceCounterKHR`.

If neither of `VK_QUERY_RESULT_WAIT_BIT` and `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` are set, results are only written out for queries in the available state.

If `VK_QUERY_RESULT_WAIT_BIT` is set, the implementation will wait for each query's status to be in the available state before retrieving the numerical results for that query. This is guaranteed to reflect

the most recent use of the query on the same queue, assuming that the query is not being simultaneously used by other queues. If the query does not become available in a finite amount of time (e.g. due to not issuing a query since the last reset), a `VK_ERROR_DEVICE_LOST` error **may** occur.

Similarly, if `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` is set and `VK_QUERY_RESULT_WAIT_BIT` is not set, the availability is guaranteed to reflect the most recent use of the query on the same queue, assuming that the query is not being simultaneously used by other queues. As with `vkGetQueryPoolResults`, implementations **must** guarantee that if they return a non-zero availability value, then the numerical results are valid.

If `VK_QUERY_RESULT_PARTIAL_BIT` is set, `VK_QUERY_RESULT_WAIT_BIT` is not set, and the query's status is unavailable, an intermediate result value between zero and the final result value is written for that query.

`VK_QUERY_RESULT_PARTIAL_BIT` **must** not be used if the pool's `queryType` is `VK_QUERY_TYPE_TIMESTAMP`.

`vkCmdCopyQueryPoolResults` is considered to be a transfer operation, and its writes to buffer memory **must** be synchronized using `VK_PIPELINE_STAGE_TRANSFER_BIT` and `VK_ACCESS_TRANSFER_WRITE_BIT` before using the results.

## Valid Usage

- `dstOffset` **must** be less than the size of `dstBuffer`
- `firstQuery` **must** be less than the number of queries in `queryPool`
- The sum of `firstQuery` and `queryCount` **must** be less than or equal to the number of queries in `queryPool`
- If `VK_QUERY_RESULT_64_BIT` is not set in `flags` then `dstOffset` and `stride` **must** be multiples of 4
- If `VK_QUERY_RESULT_64_BIT` is set in `flags` then `dstOffset` and `stride` **must** be multiples of 8
- `dstBuffer` **must** have enough storage, from `dstOffset`, to contain the result of each query, as described [here](#)
- `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TIMESTAMP`, `flags` **must** not contain `VK_QUERY_RESULT_PARTIAL_BIT`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, `VkPhysicalDevicePerformanceQueryPropertiesKHR::allowCommandBufferQueryCopies` **must** be `VK_TRUE`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, `flags` **must** not contain `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT`, `VK_QUERY_RESULT_PARTIAL_BIT` or `VK_QUERY_RESULT_64_BIT`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `queryPool` **must** have been submitted once for each pass as retrieved via a call to `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.
- `vkCmdCopyQueryPoolResults` **must** not be called if the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL`.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `dstBuffer` **must** be a valid `VkBuffer` handle
- `flags` **must** be a valid combination of `VkQueryResultFlagBits` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- Each of `commandBuffer`, `dstBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics Compute	Transfer

Rendering operations such as clears, MSAA resolves, attachment load/store operations, and blits **may** count towards the results of queries. This behavior is implementation-dependent and **may** vary depending on the path used within an implementation. For example, some implementations have several types of clears, some of which **may** include vertices and some not.

## 17.3. Occlusion Queries

Occlusion queries track the number of samples that pass the per-fragment tests for a set of drawing commands. As such, occlusion queries are only available on queue families supporting graphics operations. The application **can** then use these results to inform future rendering decisions. An occlusion query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively. When an occlusion query begins, the count of passing samples always starts at zero. For each drawing command, the count is incremented as described in [Sample Counting](#). If `flags` does not contain `VK_QUERY_CONTROL_PRECISE_BIT` an implementation **may** generate any non-zero result value for the query if the count of passing samples is non-zero.

*Note*



Not setting `VK_QUERY_CONTROL_PRECISE_BIT` mode **may** be more efficient on some implementations, and **should** be used where it is sufficient to know a boolean result on whether any samples passed the per-fragment tests. In this case, some implementations **may** only return zero or one, indifferent to the actual number of samples passing the per-fragment tests.

When an occlusion query finishes, the result for that query is marked as available. The application **can** then either copy the result to a buffer (via `vkCmdCopyQueryPoolResults`) or request it be put into host memory (via `vkGetQueryPoolResults`).

*Note*



If occluding geometry is not drawn first, samples **can** pass the depth test, but still not be visible in a final image.

## 17.4. Pipeline Statistics Queries

Pipeline statistics queries allow the application to sample a specified set of `VkPipeline` counters. These counters are accumulated by Vulkan for a set of either draw or dispatch commands while a pipeline statistics query is active. As such, pipeline statistics queries are available on queue families supporting either graphics or compute operations. The availability of pipeline statistics queries is indicated by the `pipelineStatisticsQuery` member of the `VkPhysicalDeviceFeatures` object (see `vkGetPhysicalDeviceFeatures` and `vkCreateDevice` for detecting and requesting this query type on a `VkDevice`).

A pipeline statistics query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively. When a pipeline statistics query begins, all statistics counters are set to zero. While the query is active, the pipeline type determines which set of statistics are available, but these **must** be configured on the query pool when it is created. If a statistic counter is issued on a command buffer that does not support the corresponding operation, the value of that counter is undefined after the query has finished. At least one statistic counter relevant to the operations supported on the recording command buffer **must** be enabled.

Bits which **can** be set to individually enable pipeline statistics counters for query pools with `VkQueryPoolCreateInfo::pipelineStatistics`, and for secondary command buffers with `VkCommandBufferInheritanceInfo::pipelineStatistics`, are:

```

typedef enum VkQueryPipelineStatisticFlagBits {
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT = 0x00000001,
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT = 0x00000002,
    VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT = 0x00000004,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT = 0x00000008,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT = 0x00000010,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT = 0x00000020,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT = 0x00000040,
    VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT = 0x00000080,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT = 0x00000100,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT =
        0x00000200,
    VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT = 0x00000400,
    VK_QUERY_PIPELINE_STATISTIC_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkQueryPipelineStatisticFlagBits;

```

- **VK\_QUERY\_PIPELINE\_STATISTIC\_INPUT\_ASSEMBLY\_VERTICES\_BIT** specifies that queries managed by the pool will count the number of vertices processed by the [input assembly](#) stage. Vertices corresponding to incomplete primitives **may** contribute to the count.
- **VK\_QUERY\_PIPELINE\_STATISTIC\_INPUT\_ASSEMBLY\_PRIMITIVES\_BIT** specifies that queries managed by the pool will count the number of primitives processed by the [input assembly](#) stage. If primitive restart is enabled, restarting the primitive topology has no effect on the count. Incomplete primitives **may** be counted.
- **VK\_QUERY\_PIPELINE\_STATISTIC\_VERTEX\_SHADER\_INVOCATIONS\_BIT** specifies that queries managed by the pool will count the number of vertex shader invocations. This counter's value is incremented each time a vertex shader is [invoked](#).
- **VK\_QUERY\_PIPELINE\_STATISTIC\_GEOMETRY\_SHADER\_INVOCATIONS\_BIT** specifies that queries managed by the pool will count the number of geometry shader invocations. This counter's value is incremented each time a geometry shader is [invoked](#). In the case of [instanced geometry shaders](#), the geometry shader invocations count is incremented for each separate instanced invocation.
- **VK\_QUERY\_PIPELINE\_STATISTIC\_GEOMETRY\_SHADER\_PRIMITIVES\_BIT** specifies that queries managed by the pool will count the number of primitives generated by geometry shader invocations. The counter's value is incremented each time the geometry shader emits a primitive. Restarting primitive topology using the SPIR-V instructions [OpEndPrimitive](#) or [OpEndStreamPrimitive](#) has no effect on the geometry shader output primitives count.
- **VK\_QUERY\_PIPELINE\_STATISTIC\_CLIPPING\_INVOCATIONS\_BIT** specifies that queries managed by the pool will count the number of primitives processed by the [Primitive Clipping](#) stage of the pipeline. The counter's value is incremented each time a primitive reaches the primitive clipping stage.
- **VK\_QUERY\_PIPELINE\_STATISTIC\_CLIPPING\_PRIMITIVES\_BIT** specifies that queries managed by the pool will count the number of primitives output by the [Primitive Clipping](#) stage of the pipeline. The counter's value is incremented each time a primitive passes the primitive clipping stage. The actual number of primitives output by the primitive clipping stage for a particular input primitive is implementation-dependent but **must** satisfy the following conditions:

- If at least one vertex of the input primitive lies inside the clipping volume, the counter is incremented by one or more.
- Otherwise, the counter is incremented by zero or more.
- `VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT` specifies that queries managed by the pool will count the number of fragment shader invocations. The counter’s value is incremented each time the fragment shader is [invoked](#).
- `VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT` specifies that queries managed by the pool will count the number of patches processed by the tessellation control shader. The counter’s value is incremented once for each patch for which a tessellation control shader is [invoked](#).
- `VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT` specifies that queries managed by the pool will count the number of invocations of the tessellation evaluation shader. The counter’s value is incremented each time the tessellation evaluation shader is [invoked](#).
- `VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT` specifies that queries managed by the pool will count the number of compute shader invocations. The counter’s value is incremented every time the compute shader is invoked. Implementations **may** skip the execution of certain compute shader invocations or execute additional compute shader invocations for implementation-dependent reasons as long as the results of rendering otherwise remain unchanged.

These values are intended to measure relative statistics on one implementation. Various device architectures will count these values differently. Any or all counters **may** be affected by the issues described in [Query Operation](#).



#### Note

For example, tile-based rendering devices **may** need to replay the scene multiple times, affecting some of the counts.

If a pipeline has `rasterizerDiscardEnable` enabled, implementations **may** discard primitives after the final vertex processing stage. As a result, if `rasterizerDiscardEnable` is enabled, the clipping input and output primitives counters **may** not be incremented.

When a pipeline statistics query finishes, the result for that query is marked as available. The application **can** copy the result to a buffer (via `vkCmdCopyQueryPoolResults`), or request it be put into host memory (via `vkGetQueryPoolResults`).

```
typedef VkFlags VkQueryPipelineStatisticFlags;
```

`VkQueryPipelineStatisticFlags` is a bitmask type for setting a mask of zero or more `VkQueryPipelineStatisticFlagBits`.

## 17.5. Timestamp Queries

*Timestamps* provide applications with a mechanism for timing the execution of commands. A

timestamp is an integer value generated by the `VkPhysicalDevice`. Unlike other queries, timestamps do not operate over a range, and so do not use `vkCmdBeginQuery` or `vkCmdEndQuery`. The mechanism is built around a set of commands that allow the application to tell the `VkPhysicalDevice` to write timestamp values to a `query pool` and then either read timestamp values on the host (using `vkGetQueryPoolResults`) or copy timestamp values to a `VkBuffer` (using `vkCmdCopyQueryPoolResults`). The application **can** then compute differences between timestamps to determine execution time.

The number of valid bits in a timestamp value is determined by the `VkQueueFamilyProperties::timestampValidBits` property of the queue on which the timestamp is written. Timestamps are supported on any queue which reports a non-zero value for `timestampValidBits` via `vkGetPhysicalDeviceQueueFamilyProperties`. If the `timestampComputeAndGraphics` limit is `VK_TRUE`, timestamps are supported by every queue family that supports either graphics or compute operations (see `VkQueueFamilyProperties`).

The number of nanoseconds it takes for a timestamp value to be incremented by 1 **can** be obtained from `VkPhysicalDeviceLimits::timestampPeriod` after a call to `vkGetPhysicalDeviceProperties`.

To request a timestamp, call:

```
void vkCmdWriteTimestamp(  
    VkCommandBuffer  
    VkPipelineStageFlagBits  
    VkQueryPool  
    uint32_t  
        commandBuffer,  
        pipelineStage,  
        queryPool,  
        query);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pipelineStage` is one of the `VkPipelineStageFlagBits`, specifying a stage of the pipeline.
- `queryPool` is the query pool that will manage the timestamp.
- `query` is the query within the query pool that will contain the timestamp.

`vkCmdWriteTimestamp` latches the value of the timer when all previous commands have completed executing as far as the specified pipeline stage, and writes the timestamp value to memory. When the timestamp value is written, the availability status of the query is set to available.

*Note*



If an implementation is unable to detect completion and latch the timer at any specific stage of the pipeline, it **may** instead do so at any logically later stage.

`vkCmdCopyQueryPoolResults` **can** then be called to copy the timestamp value from the query pool into buffer memory, with ordering and synchronization behavior equivalent to how other queries operate. Timestamp values **can** also be retrieved from the query pool using `vkGetQueryPoolResults`. As with other queries, the query **must** be reset using `vkCmdResetQueryPool` or `vkResetQueryPoolEXT` before requesting the timestamp value be written to it.

While `vkCmdWriteTimestamp` **can** be called inside or outside of a render pass instance, `vkCmdCopyQueryPoolResults` **must** only be called outside of a render pass instance.

Timestamps **may** only be meaningfully compared if they are written by commands submitted to the same queue.

*Note*



An example of such a comparison is determining the execution time of a sequence of commands.

If `vkCmdWriteTimestamp` is called while executing a render pass instance that has multiview enabled, the timestamp uses N consecutive query indices in the query pool (starting at `query`) where N is the number of bits set in the view mask of the subpass the command is executed in. The resulting query values are determined by an implementation-dependent choice of one of the following behaviors:

- The first query is a timestamp value and (if more than one bit is set in the view mask) zero is written to the remaining queries. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the difference between the first query written by each command.
- All N queries are timestamp values. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the sum of the difference between corresponding queries written by each command. The difference between corresponding queries **may** be the execution time of a single view.

In either case, the application **can** sum the differences between all N queries to determine the total execution time.

## Valid Usage

- `queryPool` **must** have been created with a `queryType` of `VK_QUERY_TYPE_TIMESTAMP`
- The query identified by `queryPool` and `query` **must** be *unavailable*
- The command pool's queue family **must** support a non-zero `timestampValidBits`
- All queries used by the command **must** be unavailable
- If `vkCmdWriteTimestamp` is called within a render pass instance, the sum of `query` and the number of bits set in the current subpass's view mask **must** be less than or equal to the number of queries in `queryPool`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pipelineStage` **must** be a valid `VkPipelineStageFlagBits` value
- `queryPool` **must** be a valid `VkQueryPool` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- Both of `commandBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Transfer Graphics Compute	Transfer

## 17.6. Performance Queries

*Performance queries* provide applications with a mechanism for getting performance counter information about the execution of command buffers, render passes, and commands.

Each queue family advertises the performance counters that **can** be queried on a queue of that family via a call to `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR`. Implementations **may** limit access to performance counters based on platform requirements or only to specialized drivers for development purposes.

### Note



This may include no performance counters being enumerated, or a reduced set. Please refer to platform-specific documentation for guidance on any such restrictions.

Performance queries use the existing `vkCmdBeginQuery` and `vkCmdEndQuery` to control what

command buffers, render passes, or commands to get performance information for.

Implementations **may** require multiple passes where the command buffer, render passes, or commands being recorded are the same and are executed on the same queue to record performance counter data. This is achieved by submitting the same batch and providing a [VkPerformanceQuerySubmitInfoKHR](#) structure containing a counter pass index. The number of passes required for a given performance query pool **can** be queried via a call to [vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR](#).

*Note*



Command buffers created with `enum:VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` **must** not be re-submitted. Changing command buffer usage bits **may** affect performance. To avoid this, the application **should** re-record any command buffers with the `enum:VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` when multiple counter passes are required.

Performance counter results from a performance query pool **can** be obtained with the command [vkGetQueryPoolResults](#).

Performance query results are returned in an array of [VkPerformanceCounterResultKHR](#) unions containing the data associated with each counter in the query, stored in the same order as the counters supplied in `pCounterIndices` when creating the performance query. The [VkPerformanceCounterKHR::unit](#) enumeration specifies how to parse the counter data.

```
typedef union VkPerformanceCounterResultKHR {
    int32_t    int32;
    int64_t    int64;
    uint32_t   uint32;
    uint64_t   uint64;
    float      float32;
    double     float64;
} VkPerformanceCounterResultKHR;
```

### 17.6.1. Profiling Lock

To record and submit a command buffer that contains a performance query pool the profiling lock **must** be held. The profiling lock **must** be acquired prior to any call to [vkBeginCommandBuffer](#) that will be using a performance query pool. The profiling lock **must** be held while any command buffer that contains a performance query pool is in the *recording*, *executable*, or *pending state*. To acquire the profiling lock, call:

```
VkResult vkAcquireProfilingLockKHR(
    VkDevice           device,
    const VkAcquireProfilingLockInfoKHR* pInfo);
```

- `device` is the logical device to profile.

- `pInfo` is a pointer to an instance of the `VkAcquireProfilingLockInfoKHR` structure which contains information about how the profiling is to be acquired.

Implementations **may** allow multiple actors to hold the profiling lock concurrently.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkAcquireProfilingLockInfoKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_TIMEOUT`

The `VkAcquireProfilingLockInfoKHR` structure is defined as:

```
typedef struct VkAcquireProfilingLockInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkAcquireProfilingLockFlagsKHR flags;
    uint64_t                  timeout;
} VkAcquireProfilingLockInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `timeout` indicates how long the function waits, in nanoseconds, if the profiling lock is not available.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

If `timeout` is 0, `vkAcquireProfilingLockKHR` will not block while attempting to acquire the profiling lock. If `timeout` is `UINT64_MAX`, the function will not return until the profiling lock was acquired.

```
typedef enum VkAcquireProfilingLockFlagBitsKHR {
    VK_ACQUIRE_PROFILING_LOCK_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkAcquireProfilingLockFlagBitsKHR;
```

```
typedef VkFlags VkAcquireProfilingLockFlagsKHR;
```

`VkAcquireProfilingLockFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.

To release the profiling lock, call:

```
void vkReleaseProfilingLockKHR(
    VkDevice                                     device);
```

- `device` is the logical device to cease profiling on.

### Valid Usage

- The profiling lock of `device` **must** have been held via a previous successful call to [vkAcquireProfilingLockKHR](#).

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle

## 17.7. Transform Feedback Queries

Transform feedback queries track the number of primitives attempted to be written and actually written, by the vertex stream being captured, to a transform feedback buffer. This query is updated during draw commands while transform feedback is active. The number of primitives actually written will be less than the number attempted to be written if the bound transform feedback buffer size was too small for the number of primitives actually drawn. Primitives are not written beyond the bound range of the transform feedback buffer. A transform feedback query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively to query for vertex stream zero. `vkCmdBeginQueryIndexedEXT` and `vkCmdEndQueryIndexedEXT` **can** be used to begin and end transform feedback queries for any supported vertex stream. When a transform feedback query begins, the count of primitives written and primitives needed starts from zero. For each drawing command, the count is incremented as vertex attribute outputs are captured to the transform feedback buffers while transform feedback is active.

When a transform feedback query finishes, the result for that query is marked as available. The application **can** then either copy the result to a buffer (via `vkCmdCopyQueryPoolResults`) or request it be put into host memory (via `vkGetQueryPoolResults`).

## 17.8. Intel performance queries

Intel performance queries allow an application to capture performance data for a set of commands. Performance queries are used in a similar way than other types of queries. A main difference with existing queries is that the resulting data should be handed over to a library capable of producing human readable results rather than being read directly by an application.

Prior to creating a performance query pool, initialize the device for performance queries with the call:

```
VkResult vkInitializePerformanceApiINTEL(  
    VkDevice                                     device,  
    const VkInitializePerformanceApiInfoINTEL* pInitializeInfo);
```

- `device` is the logical device used for the queries.
- `pInitializeInfo` is a pointer to a `VkInitializePerformanceApiInfoINTEL` structure specifying initialization parameters.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInitializeInfo` **must** be a valid pointer to a valid `VkInitializePerformanceApiInfoINTEL` structure

### Return Codes

#### Success

- `VK_SUCCESS`

#### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkInitializePerformanceApiInfoINTEL` structure is defined as :

```
typedef struct VkInitializePerformanceApiInfoINTEL {  
    VkStructureType    sType;  
    const void*        pNext;  
    void*              pUserData;  
} VkInitializePerformanceApiInfoINTEL;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `pUserData` is a pointer for application data.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_INITIALIZE_PERFORMANCE_API_INFO_INTEL`
- `pNext` **must** be `NULL`

Once performance query operations have completed, uninitialized the device for performance queries with the call:

```
void vkUninitializePerformanceApiINTEL(  
    VkDevice                               device);
```

- `device` is the logical device used for the queries.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle

Some performance query features of a device can be discovered with the call:

```
VkResult vkGetPerformanceParameterINTEL(  
    VkDevice                               device,  
    VkPerformanceParameterTypeINTEL      parameter,  
    VkPerformanceValueINTEL*            pValue);
```

- `device` is the logical device to query.
- `parameter` is the parameter to query.
- `pValue` is a pointer to a `VkPerformanceValueINTEL` structure in which the type and value of the parameter are returned.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `parameter` **must** be a valid `VkPerformanceParameterTypeINTEL` value
- `pValue` **must** be a valid pointer to a `VkPerformanceValueINTEL` structure

## Return Codes

### Success

- VK\_SUCCESS

### Failure

- VK\_ERROR\_TOO\_MANY\_OBJECTS
- VK\_ERROR\_OUT\_OF\_HOST\_MEMORY

Possible values of `vkGetPerformanceParameterINTEL::parameter`, specifying a performance query feature, are:

```
typedef enum VkPerformanceParameterTypeINTEL {
    VK_PERFORMANCE_PARAMETER_TYPE_HW_COUNTERS_SUPPORTED_INTEL = 0,
    VK_PERFORMANCE_PARAMETER_TYPE_STREAM_MARKER_VALID_BITS_INTEL = 1,
    VK_PERFORMANCE_PARAMETER_TYPE_MAX_ENUM_INTEL = 0x7FFFFFFF
} VkPerformanceParameterTypeINTEL;
```

- `VK_PERFORMANCE_PARAMETER_TYPE_HW_COUNTERS_SUPPORTED_INTEL` has a boolean result which tells whether hardware counters can be captured.
- `VK_PERFORMANCE_PARAMETER_TYPE_STREAM_MARKER_VALID_BITS_INTEL` has a 32 bits integer result which tells how many bits can be written into the `VkPerformanceValueINTEL` value.

The `VkPerformanceValueINTEL` structure is defined as:

```
typedef struct VkPerformanceValueINTEL {
    VkPerformanceValueTypeINTEL    type;
    VkPerformanceValueDataINTEL   data;
} VkPerformanceValueINTEL;
```

- `type` is a `VkPerformanceValueTypeINTEL` value specifying the type of the returned data.
- `data` is a `VkPerformanceValueDataINTEL` union specifying the value of the returned data.

## Valid Usage (Implicit)

- `type` **must** be a valid `VkPerformanceValueTypeINTEL` value
- `data` **must** be a valid `VkPerformanceValueDataINTEL` union

Possible values of `VkPerformanceValueINTEL::type`, specifying the type of the data returned in `VkPerformanceValueINTEL::data`, are:

- `VK_PERFORMANCE_VALUE_TYPE_UINT32_INTEL` specifies that unsigned 32-bit integer data is returned in `data.value32`.
- `VK_PERFORMANCE_VALUE_TYPE_UINT64_INTEL` specifies that unsigned 64-bit integer data is returned

in `data.value64`.

- `VK_PERFORMANCE_VALUE_TYPE_FLOAT_INTEL` specifies that floating-point data is returned in `data.valueFloat`.
- `VK_PERFORMANCE_VALUE_TYPE_BOOL_INTEL` specifies that `Bool32` data is returned in `data.valueBool`.
- `VK_PERFORMANCE_VALUE_TYPE_STRING_INTEL` specifies that a pointer to a null-terminated UTF-8 string is returned in `data.valueString`. The pointer is valid for the lifetime of the `device` parameter passed to `vkGetPerformanceParameterINTEL`.

```
typedef enum VkPerformanceValueTypeINTEL {
    VK_PERFORMANCE_VALUE_TYPE_UINT32_INTEL = 0,
    VK_PERFORMANCE_VALUE_TYPE_UINT64_INTEL = 1,
    VK_PERFORMANCE_VALUE_TYPE_FLOAT_INTEL = 2,
    VK_PERFORMANCE_VALUE_TYPE_BOOL_INTEL = 3,
    VK_PERFORMANCE_VALUE_TYPE_STRING_INTEL = 4,
    VK_PERFORMANCE_VALUE_TYPE_MAX_ENUM_INTEL = 0x7FFFFFFF
} VkPerformanceValueTypeINTEL;
```

The `VkPerformanceValueDataINTEL` union is defined as:

```
typedef union VkPerformanceValueDataINTEL {
    uint32_t      value32;
    uint64_t      value64;
    float         valueFloat;
    VkBool32      valueBool;
    const char*   valueString;
} VkPerformanceValueDataINTEL;
```

- `data.value32` represents 32-bit integer data.
- `data.value64` represents 64-bit integer data.
- `data.valueFloat` represents floating-point data.
- `data.valueBool` represents `Bool32` data.
- `data.valueString` represents a pointer to a null-terminated UTF-8 string.

The correct member of the union is determined by the associated `VkPerformanceValueTypeINTEL` value.

### Valid Usage (Implicit)

- `valueString` must be a valid pointer to a valid

The `VkQueryPoolCreateInfoINTEL` structure is defined as:

```

typedef struct VkQueryPoolCreateInfoINTEL {
    VkStructureType           sType;
    const void*              pNext;
    VkQueryPoolSamplingModeINTEL performanceCountersSampling;
} VkQueryPoolCreateInfoINTEL;

```

To create a pool for Intel performance queries, set `VkQueryPoolCreateInfo::queryType` to `VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL` and add a `VkQueryPoolCreateInfoINTEL` to the `pNext` chain of the `VkQueryPoolCreateInfo` structure.

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `performanceCountersSampling` describe how performance queries should be captured.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO_INTEL`
- `pNext` **must** be `NULL`
- `performanceCountersSampling` **must** be a valid `VkQueryPoolSamplingModeINTEL` value

Possible values of `VkQueryPoolCreateInfoINTEL::performanceCountersSampling` are:

```

typedef enum VkQueryPoolSamplingModeINTEL {
    VK_QUERY_POOL_SAMPLING_MODE_MANUAL_INTEL = 0,
    VK_QUERY_POOL_SAMPLING_MODE_MAX_ENUM_INTEL = 0x7FFFFFFF
} VkQueryPoolSamplingModeINTEL;

```

- `VK_QUERY_POOL_SAMPLING_MODE_MANUAL_INTEL` is the default mode in which the application calls `vkCmdBeginQuery` and `vkCmdEndQuery` to record performance data.

To help associate query results with a particular point at which an application emitted commands, markers can be set into the command buffers with the call:

```

VkResult vkCmdSetPerformanceMarkerINTEL(
    VkCommandBuffer                      commandBuffer,
    const VkPerformanceMarkerInfoINTEL* pMarkerInfo);

```

The last marker set onto a command buffer before the end of a query will be part of the query result.

## Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `pMarkerInfo` must be a valid pointer to a valid `VkPerformanceMarkerInfoINTEL` structure
- `commandBuffer` must be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, compute, or transfer operations

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute Transfer	

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkPerformanceMarkerInfoINTEL` structure is defined as:

```
typedef struct VkPerformanceMarkerInfoINTEL {
    VkStructureType    sType;
    const void*        pNext;
    uint64_t           marker;
} VkPerformanceMarkerInfoINTEL;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `marker` is the marker value that will be recorded into the opaque query results.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PERFORMANCE_MARKER_INFO_INTEL`
- `pNext` **must** be `NULL`

When monitoring the behavior of an application within the dataset generated by the entire set of applications running on the system, it is useful to identify draw calls within a potentially huge amount of performance data. To do so, application can generate stream markers that will be used to trace back a particular draw call with a particular performance data item.

```
VkResult vkCmdSetPerformanceStreamMarkerINTEL(  
    VkCommandBuffer                           commandBuffer,  
    const VkPerformanceStreamMarkerInfoINTEL* pMarkerInfo);
```

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pMarkerInfo` **must** be a valid pointer to a valid `VkPerformanceStreamMarkerInfoINTEL` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, compute, or transfer operations

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute Transfer	

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkPerformanceStreamMarkerInfoINTEL` structure is defined as:

```
typedef struct VkPerformanceStreamMarkerInfoINTEL {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           marker;
} VkPerformanceStreamMarkerInfoINTEL;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `marker` is the marker value that will be recorded into the reports consumed by an external application.

## Valid Usage

- The value written by the application into `marker` **must** only used the valid bits as reported by `vkGetPerformanceParameterINTEL` with the `VK_PERFORMANCE_PARAMETER_TYPE_STREAM_MARKER_VALID_BITS_INTEL`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PERFORMANCE_STREAM_MARKER_INFO_INTEL`
- `pNext` **must** be `NULL`

Some applications might want measure the effect of a set of commands with a different settings. It is possible to override a particular settings using :

```
VkResult vkCmdSetPerformanceOverrideINTEL(
    VkCommandBuffer          commandBuffer,
    const VkPerformanceOverrideInfoINTEL* pOverrideInfo);
```

- `commandBuffer` is the command buffer where the override takes place.
- `pOverrideInfo` is a pointer to a `VkPerformanceOverrideInfoINTEL` structure selecting the

parameter to override.

## Valid Usage

- `pOverrideInfo` **must** not be used with a `VkPerformanceOverrideTypeINTEL` that is not reported available by `vkGetPerformanceParameterINTEL`.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pOverrideInfo` **must** be a valid pointer to a valid `VkPerformanceOverrideInfoINTEL` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, compute, or transfer operations

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute Transfer	

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkPerformanceOverrideInfoINTEL` structure is defined as:

```

typedef struct VkPerformanceOverrideInfoINTEL {
    VkStructureType          sType;
    const void*             pNext;
    VkPerformanceOverrideTypeINTEL type;
    VkBool32                 enable;
    uint64_t                  parameter;
} VkPerformanceOverrideInfoINTEL;

```

- **type** is the particular `VkPerformanceOverrideTypeINTEL` to set.
- **enable** defines whether the override is enabled.
- **parameter** is a potential required parameter for the override.

## Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_PERFORMANCE_OVERRIDE_INFO_INTEL`
- **pNext** **must** be `NULL`
- **type** **must** be a valid `VkPerformanceOverrideTypeINTEL` value

Possible values of `VkPerformanceOverrideInfoINTEL::type`, specifying performance override types, are:

```

typedef enum VkPerformanceOverrideTypeINTEL {
    VK_PERFORMANCE_OVERRIDE_TYPE_NULL_HARDWARE_INTEL = 0,
    VK_PERFORMANCE_OVERRIDE_TYPE_FLUSH_GPU_CACHES_INTEL = 1,
    VK_PERFORMANCE_OVERRIDE_TYPE_MAX_ENUM_INTEL = 0x7FFFFFFF
} VkPerformanceOverrideTypeINTEL;

```

- `VK_PERFORMANCE_OVERRIDE_TYPE_NULL_HARDWARE_INTEL` turns all rendering operations into noop.
- `VK_PERFORMANCE_OVERRIDE_TYPE_FLUSH_GPU_CACHES_INTEL` stalls the stream of commands until all previously emitted commands have completed and all caches been flushed and invalidated.

Before submitting command buffers containing performance queries commands to a device queue, the application must acquire and set a performance query configuration. The configuration can be released once all command buffers containing performance query commands are not in a pending state.

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPerformanceConfigurationINTEL)
```

To acquire a device performance configuration, call:

```
VkResult vkAcquirePerformanceConfigurationINTEL(  
    VkDevice device,  
    const VkPerformanceConfigurationAcquireInfoINTEL* pAcquireInfo,  
    VkPerformanceConfigurationINTEL* pConfiguration);
```

- **device** is the logical device that the performance query commands will be submitted to.
- **pAcquireInfo** is a pointer to a `VkPerformanceConfigurationAcquireInfoINTEL` structure, specifying the performance configuration to acquire.
- **pConfiguration** is a pointer to a `VkPerformanceConfigurationINTEL` handle in which the resulting configuration object is returned.

## Valid Usage (Implicit)

- **device** **must** be a valid `VkDevice` handle
- **pAcquireInfo** **must** be a valid pointer to a `VkPerformanceConfigurationAcquireInfoINTEL` structure
- **pConfiguration** **must** be a valid pointer to a `VkPerformanceConfigurationINTEL` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkPerformanceConfigurationAcquireInfoINTEL` structure is defined as:

```
typedef struct VkPerformanceConfigurationAcquireInfoINTEL {  
    VkStructureType sType;  
    const void* pNext;  
    VkPerformanceConfigurationTypeINTEL type;  
} VkPerformanceConfigurationAcquireInfoINTEL;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **type** is one of the `VkPerformanceConfigurationTypeINTEL` type of performance configuration that will be acquired.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PERFORMANCE_CONFIGURATION_ACQUIRE_INFO_INTEL`
- `pNext` **must** be `NULL`
- `type` **must** be a valid `VkPerformanceConfigurationTypeINTEL` value

Possible values of `VkPerformanceConfigurationAcquireInfoINTEL::type`, specifying performance configuration types, are:

```
typedef enum VkPerformanceConfigurationTypeINTEL {
    VK_PERFORMANCE_CONFIGURATION_TYPE_COMMAND_QUEUE_METRICS_DISCOVERY_ACTIVATED_INTEL
= 0,
    VK_PERFORMANCE_CONFIGURATION_TYPE_MAX_ENUM_INTEL = 0x7FFFFFFF
} VkPerformanceConfigurationTypeINTEL;
```

To set a performance configuration, call:

```
VkResult vkQueueSetPerformanceConfigurationINTEL(
    VkQueue                           queue,
    VkPerformanceConfigurationINTEL   configuration);
```

- `queue` is the queue on which the configuration will be used.
- `configuration` is the configuration to use.

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- `configuration` **must** be a valid `VkPerformanceConfigurationINTEL` handle
- Both of `configuration`, and `queue` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	Any	-

## Return Codes

### Success

- VK\_SUCCESS

### Failure

- VK\_ERROR\_TOO\_MANY\_OBJECTS
- VK\_ERROR\_OUT\_OF\_HOST\_MEMORY

To release a device performance configuration, call:

```
VkResult vkReleasePerformanceConfigurationINTEL(  
    VkDevice           device,  
    VkPerformanceConfigurationINTEL configuration);
```

- **device** is the device associated to the configuration object to release.
- **configuration** is the configuration object to release.

## Valid Usage

- **configuration must** not be released before all command buffers submitted while the configuration was set are in **pending state**.

## Valid Usage (Implicit)

- **device must** be a valid **VkDevice** handle
- **configuration must** be a valid **VkPerformanceConfigurationINTEL** handle
- **configuration must** have been created, allocated, or retrieved from **device**

## Return Codes

### Success

- VK\_SUCCESS

### Failure

- VK\_ERROR\_TOO\_MANY\_OBJECTS
- VK\_ERROR\_OUT\_OF\_HOST\_MEMORY

# Chapter 18. Clear Commands

## 18.1. Clearing Images Outside A Render Pass Instance

Color and depth/stencil images **can** be cleared outside a render pass instance using [vkCmdClearColorImage](#) or [vkCmdClearDepthStencilImage](#), respectively. These commands are only allowed outside of a render pass instance.

To clear one or more subranges of a color image, call:

```
void vkCmdClearColorImage(  
    VkCommandBuffer  
    VkImage  
    VkImageLayout  
    const VkClearColorValue*  
    uint32_t  
    const VkImageSubresourceRange*  
                                commandBuffer,  
                                image,  
                                imageLayout,  
                                pColor,  
                                rangeCount,  
                                pRanges);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **image** is the image to be cleared.
- **imageLayout** specifies the current layout of the image subresource ranges to be cleared, and **must** be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.
- **pColor** is a pointer to a [VkClearColorValue](#) structure containing the values that the image subresource ranges will be cleared to (see [Clear Values](#) below).
- **rangeCount** is the number of image subresource range structures in **pRanges**.
- **pRanges** is a pointer to an array of [VkImageSubresourceRange](#) structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in [Image Views](#).

Each specified range in **pRanges** is cleared to the value specified by **pColor**.

## Valid Usage

- The `format features` of `image` **must** contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.
- `image` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- `image` **must** not use a format listed in `Formats requiring sampler Y'CBCR conversion for VK_IMAGE_ASPECT_COLOR_BIT image views`
- If `image` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `imageLayout` **must** specify the layout of the image subresource ranges of `image` specified in `pRanges` at the time this command is executed on a `VkDevice`
- `imageLayout` **must** be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
- The `VkImageSubresourceRange::aspectMask` members of the elements of the `pRanges` array **must** each only include `VK_IMAGE_ASPECT_COLOR_BIT`
- The `VkImageSubresourceRange::baseMipLevel` members of the elements of the `pRanges` array **must** each be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- For each `VkImageSubresourceRange` element of `pRanges`, if the `levelCount` member is not `VK_REMAINING_MIP_LEVELS`, then `baseMipLevel + levelCount` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- The `VkImageSubresourceRange::baseArrayLayer` members of the elements of the `pRanges` array **must** each be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- For each `VkImageSubresourceRange` element of `pRanges`, if the `layerCount` member is not `VK_REMAINING_ARRAY_LAYERS`, then `baseArrayLayer + layerCount` **must** be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- `image` **must** not have a compressed or depth/stencil format
- If `commandBuffer` is an unprotected command buffer, then `image` **must** not be a protected image
- If `commandBuffer` is a protected command buffer, then `image` **must** not be an unprotected image

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `image` **must** be a valid `VkImage` handle
- `imageLayout` **must** be a valid `VkImageLayout` value
- `pColor` **must** be a valid pointer to a valid `VkClearColorValue` union
- `pRanges` **must** be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- `rangeCount` **must** be greater than `0`
- Both of `commandBuffer`, and `image` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics Compute	Transfer

To clear one or more subranges of a depth/stencil image, call:

```
void vkCmdClearDepthStencilImage(  
    VkCommandBuffer  
    VkImage  
    VkImageLayout  
    const VkClearDepthStencilValue*  
    uint32_t  
    const VkImageSubresourceRange*  
        commandBuffer,  
        image,  
        imageLayout,  
        pDepthStencil,  
        rangeCount,  
        pRanges);
```

- `commandBuffer` is the command buffer into which the command will be recorded.

- `image` is the image to be cleared.
- `imageLayout` specifies the current layout of the image subresource ranges to be cleared, and **must** be `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.
- `pDepthStencil` is a pointer to a `VkClearDepthStencilValue` structure containing the values that the depth and stencil image subresource ranges will be cleared to (see [Clear Values](#) below).
- `rangeCount` is the number of image subresource range structures in `pRanges`.
- `pRanges` is a pointer to an array of `VkImageSubresourceRange` structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in [Image Views](#).

## Valid Usage

- The `format` features of `image` **must** contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.
- If any element of `pRanges.aspect` includes `VK_IMAGE_ASPECT_STENCIL_BIT`, and `image` was created with `separate stencil usage`, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` **must** have been included in the `VkImageStencilUsageCreateInfoEXT::stencilUsage` used to create `image`
- If any element of `pRanges.aspect` includes `VK_IMAGE_ASPECT_STENCIL_BIT`, and `image` was not created with `separate stencil usage`, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` **must** have been included in the `VkImageCreateInfo::usage` used to create `image`
- If any element of `pRanges.aspect` includes `VK_IMAGE_ASPECT_DEPTH_BIT`, `VK_IMAGE_USAGE_TRANSFER_DST_BIT` **must** have been included in the `VkImageCreateInfo::usage` used to create `image`
- If `image` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `imageLayout` **must** specify the layout of the image subresource ranges of `image` specified in `pRanges` at the time this command is executed on a `VkDevice`
- `imageLayout` **must** be either of `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- The `VkImageSubresourceRange::aspectMask` members of the elements of the `pRanges` array **must** each only include `VK_IMAGE_ASPECT_DEPTH_BIT` if the image format has a depth component
- The `VkImageSubresourceRange::aspectMask` members of the elements of the `pRanges` array **must** each only include `VK_IMAGE_ASPECT_STENCIL_BIT` if the image format has a stencil component
- The `VkImageSubresourceRange::baseMipLevel` members of the elements of the `pRanges` array **must** each be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- For each `VkImageSubresourceRange` element of `pRanges`, if the `levelCount` member is not `VK_REMAINING_MIP_LEVELS`, then `baseMipLevel + levelCount` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- The `VkImageSubresourceRange::baseArrayLayer` members of the elements of the `pRanges` array **must** each be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- For each `VkImageSubresourceRange` element of `pRanges`, if the `layerCount` member is not `VK_REMAINING_ARRAY_LAYERS`, then `baseArrayLayer + layerCount` **must** be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created
- `image` **must** have a depth/stencil format
- If `commandBuffer` is an unprotected command buffer, then `image` **must** not be a protected image
- If `commandBuffer` is a protected command buffer, then `image` **must** not be an unprotected image

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `image` **must** be a valid `VkImage` handle
- `imageLayout` **must** be a valid `VkImageLayout` value
- `pDepthStencil` **must** be a valid pointer to a valid `VkClearDepthStencilValue` structure
- `pRanges` **must** be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called outside of a render pass instance
- `rangeCount` **must** be greater than `0`
- Both of `commandBuffer`, and `image` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics	Transfer

Clears outside render pass instances are treated as transfer operations for the purposes of memory barriers.

## 18.2. Clearing Images Inside A Render Pass Instance

To clear one or more regions of color and depth/stencil attachments inside a render pass instance, call:

```
void vkCmdClearAttachments(  
    VkCommandBuffer  
    uint32_t  
    const VkClearAttachment*  
    uint32_t  
    const VkClearRect*  
                                commandBuffer,  
                                attachmentCount,  
                                pAttachments,  
                                rectCount,  
                                pRects);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `attachmentCount` is the number of entries in the `pAttachments` array.
- `pAttachments` is a pointer to an array of `VkClearAttachment` structures defining the attachments to clear and the clear values to use. If any attachment to be cleared in the current subpass is `VK_ATTACHMENT_UNUSED`, then the clear has no effect on that attachment.
- `rectCount` is the number of entries in the `pRects` array.
- `pRects` is a pointer to an array of `VkClearRect` structures defining regions within each selected attachment to clear.

`vkCmdClearAttachments` can clear multiple regions of each attachment used in the current subpass of a render pass instance. This command must be called only inside a render pass instance, and implicitly selects the images to clear based on the current framebuffer attachments and the command parameters.

If the render pass has a [fragment density map attachment](#), clears follow the [operations of fragment density maps](#) as if each clear region was a primitive which generates fragments. The clear color is applied to all pixels inside each fragment's area regardless if the pixels lie outside of the clear region. Clears may have a different set of supported fragment areas than draws.

Unlike other [clear commands](#), `vkCmdClearAttachments` executes as a drawing command, rather than a transfer command, with writes performed by it executing in [rasterization order](#). Clears to color attachments are executed as color attachment writes, by the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` stage. Clears to depth/stencil attachments are executed as [depth writes](#) and [writes](#) by the `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT` and `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT` stages.

## Valid Usage

- If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_COLOR_BIT`, then the `colorAttachment` member of that element **must** either refer to a color attachment which is `VK_ATTACHMENT_UNUSED`, or **must** be a valid color attachment.
- If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_DEPTH_BIT`, then the current subpass' depth/stencil attachment **must** either be `VK_ATTACHMENT_UNUSED`, or **must** have a depth component
- If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_STENCIL_BIT`, then the current subpass' depth/stencil attachment **must** either be `VK_ATTACHMENT_UNUSED`, or **must** have a stencil component
- The `rect` member of each element of `pRects` **must** have an `extent.width` greater than `0`
- The `rect` member of each element of `pRects` **must** have an `extent.height` greater than `0`
- The rectangular region specified by each element of `pRects` **must** be contained within the render area of the current render pass instance
- The layers specified by each element of `pRects` **must** be contained within every attachment that `pAttachments` refers to
- The `layerCount` member of each element of `pRects` **must** not be `0`
- If `commandBuffer` is an unprotected command buffer, then each attachment to be cleared **must** not be a protected image.
- If `commandBuffer` is a protected command buffer, then each attachment to be cleared **must** not be an unprotected image.
- If the render pass instance this is recorded in uses multiview, then `baseArrayLayer` **must** be zero and `layerCount` **must** be one.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkClearAttachment` structures
- `pRects` **must** be a valid pointer to an array of `rectCount` `VkClearRect` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- `attachmentCount` **must** be greater than `0`
- `rectCount` **must** be greater than `0`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

The `VkClearRect` structure is defined as:

```
typedef struct VkClearRect {  
    VkRect2D      rect;  
    uint32_t      baseArrayLayer;  
    uint32_t      layerCount;  
} VkClearRect;
```

- `rect` is the two-dimensional region to be cleared.
- `baseArrayLayer` is the first layer to be cleared.
- `layerCount` is the number of layers to clear.

The layers `[baseArrayLayer, baseArrayLayer + layerCount)` counting from the base layer of the attachment image view are cleared.

The `VkClearAttachment` structure is defined as:

```
typedef struct VkClearAttachment {  
    VkImageAspectFlags aspectMask;  
    uint32_t          colorAttachment;  
    VkClearValue      clearValue;  
} VkClearAttachment;
```

- `aspectMask` is a mask selecting the color, depth and/or stencil aspects of the attachment to be cleared.
- `colorAttachment` is only meaningful if `VK_IMAGE_ASPECT_COLOR_BIT` is set in `aspectMask`, in which case it is an index to the `pColorAttachments` array in the `VkSubpassDescription` structure of the current subpass which selects the color attachment to clear.
- `clearValue` is the color or depth/stencil value to clear the attachment to, as described in [Clear](#)

Values below.

No memory barriers are needed between `vkCmdClearAttachments` and preceding or subsequent draw or attachment clear commands in the same subpass.

The `vkCmdClearAttachments` command is not affected by the bound pipeline state.

Attachments **can** also be cleared at the beginning of a render pass instance by setting `loadOp` (or `stencilLoadOp`) of `VkAttachmentDescription` to `VK_ATTACHMENT_LOAD_OP_CLEAR`, as described for `vkCreateRenderPass`.

### Valid Usage

- If `aspectMask` includes `VK_IMAGE_ASPECT_COLOR_BIT`, it **must** not include `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`
- `aspectMask` **must** not include `VK_IMAGE_ASPECT_METADATA_BIT`
- `aspectMask` **must** not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`.
- `clearValue` **must** be a valid `VkClearColorValue` union

### Valid Usage (Implicit)

- `aspectMask` **must** be a valid combination of `VkImageAspectFlagBits` values
- `aspectMask` **must** not be `0`

## 18.3. Clear Values

The `VkClearColorValue` structure is defined as:

```
typedef union VkClearColorValue {
    float      float32[4];
    int32_t    int32[4];
    uint32_t   uint32[4];
} VkClearColorValue;
```

- `float32` are the color clear values when the format of the image or attachment is one of the formats in the [Interpretation of Numeric Format](#) table other than signed integer (`SINT`) or unsigned integer (`UINT`). Floating point values are automatically converted to the format of the image, with the clear value being treated as linear if the image is sRGB.
- `int32` are the color clear values when the format of the image or attachment is signed integer (`SINT`). Signed integer values are converted to the format of the image by casting to the smaller type (with negative 32-bit values mapping to negative values in the smaller type). If the integer clear value is not representable in the target type (e.g. would overflow in conversion to that type), the clear value is undefined.

- `uint32` are the color clear values when the format of the image or attachment is unsigned integer (`UINT`). Unsigned integer values are converted to the format of the image by casting to the integer type with fewer bits.

The four array elements of the clear color map to R, G, B, and A components of image formats, in order.

If the image has more than one sample, the same value is written to all samples for any pixels being cleared.

The `VkClearDepthStencilValue` structure is defined as:

```
typedef struct VkClearDepthStencilValue {
    float      depth;
    uint32_t   stencil;
} VkClearDepthStencilValue;
```

- `depth` is the clear value for the depth aspect of the depth/stencil attachment. It is a floating-point value which is automatically converted to the attachment's format.
- `stencil` is the clear value for the stencil aspect of the depth/stencil attachment. It is a 32-bit integer value which is converted to the attachment's format by taking the appropriate number of LSBs.

### Valid Usage

- Unless the `VK_EXT_depth_range_unrestricted` extension is enabled `depth` **must** be between `0.0` and `1.0`, inclusive

The `VkClearValue` union is defined as:

```
typedef union VkClearValue {
    VkClearColorValue        color;
    VkClearDepthStencilValue depthStencil;
} VkClearValue;
```

- `color` specifies the color image clear values to use when clearing a color image or attachment.
- `depthStencil` specifies the depth and stencil clear values to use when clearing a depth/stencil image or attachment.

This union is used where part of the API requires either color or depth/stencil clear values, depending on the attachment, and defines the initial clear values in the `VkRenderPassBeginInfo` structure.

## 18.4. Filling Buffers

To clear buffer data, call:

```
void vkCmdFillBuffer(  
    VkCommandBuffer           commandBuffer,  
    VkBuffer                  dstBuffer,  
    VkDeviceSize              dstOffset,  
    VkDeviceSize              size,  
    uint32_t                 data);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `dstBuffer` is the buffer to be filled.
- `dstOffset` is the byte offset into the buffer at which to start filling, and **must** be a multiple of 4.
- `size` is the number of bytes to fill, and **must** be either a multiple of 4, or `VK_WHOLE_SIZE` to fill the range from `offset` to the end of the buffer. If `VK_WHOLE_SIZE` is used and the remaining size of the buffer is not a multiple of 4, then the nearest smaller multiple is used.
- `data` is the 4-byte word written repeatedly to the buffer to fill `size` bytes of data. The data word is written to memory according to the host endianness.

`vkCmdFillBuffer` is treated as “transfer” operation for the purposes of synchronization barriers. The `VK_BUFFER_USAGE_TRANSFER_DST_BIT` **must** be specified in `usage` of `VkBufferCreateInfo` in order for the buffer to be compatible with `vkCmdFillBuffer`.

### Valid Usage

- `dstOffset` **must** be less than the size of `dstBuffer`
- `dstOffset` **must** be a multiple of 4
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** be greater than 0
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** be less than or equal to the size of `dstBuffer` minus `dstOffset`
- If `size` is not equal to `VK_WHOLE_SIZE`, `size` **must** be a multiple of 4
- `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- If `commandBuffer` is an unprotected command buffer, then `dstBuffer` **must** not be a protected buffer
- If `commandBuffer` is a protected command buffer, then `dstBuffer` **must** not be an unprotected buffer

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `dstBuffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics or compute operations
- This command **must** only be called outside of a render pass instance
- Both of `commandBuffer`, and `dstBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Transfer Graphics Compute	Transfer

## 18.5. Updating Buffers

To update buffer data inline in a command buffer, call:

```
void vkCmdUpdateBuffer(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
    VkDeviceSize  
    const void*
```

```
                                commandBuffer,  
                                dstBuffer,  
                                dstOffset,  
                                dataSize,  
                                pData);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `dstBuffer` is a handle to the buffer to be updated.
- `dstOffset` is the byte offset into the buffer to start updating, and **must** be a multiple of 4.
- `dataSize` is the number of bytes to update, and **must** be a multiple of 4.

- `pData` is a pointer to the source data for the buffer update, and **must** be at least `dataSize` bytes in size.

`dataSize` **must** be less than or equal to 65536 bytes. For larger updates, applications **can** use buffer to buffer [copies](#).

*Note*

Buffer updates performed with `vkCmdUpdateBuffer` first copy the data into command buffer memory when the command is recorded (which requires additional storage and may incur an additional allocation), and then copy the data from the command buffer into `dstBuffer` when the command is executed on a device.



The additional cost of this functionality compared to [buffer to buffer copies](#) means it is only recommended for very small amounts of data, and is why it is limited to only 65536 bytes.

Applications **can** work around this by issuing multiple `vkCmdUpdateBuffer` commands to different ranges of the same buffer, but it is strongly recommended that they **should not**.

The source data is copied from the user pointer to the command buffer when the command is called.

`vkCmdUpdateBuffer` is only allowed outside of a render pass. This command is treated as “transfer” operation, for the purposes of synchronization barriers. The `VK_BUFFER_USAGE_TRANSFER_DST_BIT` **must** be specified in `usage` of `VkBufferCreateInfo` in order for the buffer to be compatible with `vkCmdUpdateBuffer`.

## Valid Usage

- `dstOffset` **must** be less than the size of `dstBuffer`
- `dataSize` **must** be less than or equal to the size of `dstBuffer` minus `dstOffset`
- `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstOffset` **must** be a multiple of 4
- `dataSize` **must** be less than or equal to 65536
- `dataSize` **must** be a multiple of 4
- If `commandBuffer` is an unprotected command buffer, then `dstBuffer` **must** not be a protected buffer
- If `commandBuffer` is a protected command buffer, then `dstBuffer` **must** not be an unprotected buffer

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `dstBuffer` **must** be a valid `VkBuffer` handle
- `pData` **must** be a valid pointer to an array of `dataSize` bytes
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- `dataSize` **must** be greater than `0`
- Both of `commandBuffer`, and `dstBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Transfer Graphics Compute	Transfer

### Note



The `pData` parameter was of type `uint32_t*` instead of `void*` prior to version 1.0.19 of the Specification and `VK_HEADER_VERSION` 19 of the [Vulkan Header Files](#). This was a historical anomaly, as the source data may be of other types.

# Chapter 19. Copy Commands

An application **can** copy buffer and image data using several methods depending on the type of data transfer. Data **can** be copied between buffer objects with `vkCmdCopyBuffer` and a portion of an image **can** be copied to another image with `vkCmdCopyImage`. Image data **can** also be copied to and from buffer memory using `vkCmdCopyImageToBuffer` and `vkCmdCopyBufferToImage`. Image data **can** be blitted (with or without scaling and filtering) with `vkCmdBlitImage`. Multisampled images **can** be resolved to a non-multisampled image with `vkCmdResolveImage`.

## 19.1. Common Operation

The following valid usage rules apply to all copy commands:

- Copy commands **must** be recorded outside of a render pass instance.
- The set of all bytes bound to all the source regions **must** not overlap the set of all bytes bound to the destination regions.
- The set of all bytes bound to each destination region **must** not overlap the set of all bytes bound to another destination region.
- Copy regions **must** be non-empty.
- Regions **must** not extend outside the bounds of the buffer or image level, except that regions of compressed images **can** extend as far as the dimension of the image level rounded up to a complete compressed texel block.
- Source image subresources **must** be in either the `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` layout. Destination image subresources **must** be in the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` layout. As a consequence, if an image subresource is used as both source and destination of a copy, it **must** be in the `VK_IMAGE_LAYOUT_GENERAL` layout.
- Source images **must** have `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT` in their **format features**.
- Destination images **must** have `VK_FORMAT_FEATURE_TRANSFER_DST_BIT` in their **format features**.
- Source buffers **must** have been created with the `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage bit enabled and destination buffers **must** have been created with the `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage bit enabled.
- If the stencil aspect of source image is accessed, and the source image was not created with **separate stencil usage**, the source image **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` set in `VkImageCreateInfo::usage`
- If the stencil aspect of destination image is accessed, and the destination image was not created with **separate stencil usage**, the destination image **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` set in `VkImageCreateInfo::usage`
- If the stencil aspect of source image is accessed, and the source image was created with **separate stencil usage**, the source image **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` set in `VkImageStencilUsageCreateInfoEXT::stencilUsage`
- If the stencil aspect of destination image is accessed, and the destination image was created

with `separate stencil usage`, the destination image **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` set in `VkImageStencilUsageCreateInfoEXT::stencilUsage`

- If non-stencil aspects of a source image are accessed, the source image **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` set in `VkImageCreateInfo::usage`
- If non-stencil aspects of a source image are accessed, the source image **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` set in `VkImageCreateInfo::usage`

All copy commands are treated as “transfer” operations for the purposes of synchronization barriers.

## 19.2. Copying Data Between Buffers

To copy data between buffer objects, call:

```
void vkCmdCopyBuffer(  
    VkCommandBuffer  
    VkBuffer  
    VkBuffer  
    uint32_t  
    const VkBufferCopy*
```

`commandBuffer,`  
`srcBuffer,`  
`dstBuffer,`  
`regionCount,`  
`pRegions);`

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcBuffer` is the source buffer.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferCopy` structures specifying the regions to copy.

Each region in `pRegions` is copied from the source buffer to the same region of the destination buffer. `srcBuffer` and `dstBuffer` **can** be the same buffer or alias the same memory, but the resulting values are undefined if the copy regions overlap in memory.

## Valid Usage

- The `srcOffset` member of each element of `pRegions` **must** be less than the size of `srcBuffer`
- The `dstOffset` member of each element of `pRegions` **must** be less than the size of `dstBuffer`
- The `size` member of each element of `pRegions` **must** be less than or equal to the size of `srcBuffer` minus `srcOffset`
- The `size` member of each element of `pRegions` **must** be less than or equal to the size of `dstBuffer` minus `dstOffset`
- The union of the source regions, and the union of the destination regions, specified by the elements of `pRegions`, **must** not overlap in memory
- `srcBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag
- If `srcBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- If `commandBuffer` is an unprotected command buffer, then `srcBuffer` **must** not be a protected buffer
- If `commandBuffer` is an unprotected command buffer, then `dstBuffer` **must** not be a protected buffer
- If `commandBuffer` is a protected command buffer, then `dstBuffer` **must** not be an unprotected buffer

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `srcBuffer` **must** be a valid `VkBuffer` handle
- `dstBuffer` **must** be a valid `VkBuffer` handle
- `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkBufferCopy` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- `regionCount` **must** be greater than 0
- Each of `commandBuffer`, `dstBuffer`, and `srcBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Transfer Graphics Compute	Transfer

The `VkBufferCopy` structure is defined as:

```
typedef struct VkBufferCopy {
    VkDeviceSize    srcOffset;
    VkDeviceSize    dstOffset;
    VkDeviceSize    size;
} VkBufferCopy;
```

- `srcOffset` is the starting offset in bytes from the start of `srcBuffer`.
- `dstOffset` is the starting offset in bytes from the start of `dstBuffer`.
- `size` is the number of bytes to copy.

## Valid Usage

- The `size` **must** be greater than `0`

## 19.3. Copying Data Between Images

`vkCmdCopyImage` performs image copies in a similar manner to a host `memcpy`. It does not perform general-purpose conversions such as scaling, resizing, blending, color-space conversion, or format conversions. Rather, it simply copies raw image data. `vkCmdCopyImage` **can** copy between images with different formats, provided the formats are compatible as defined below.

To copy data between image objects, call:

```

void vkCmdCopyImage(
    VkCommandBuffer
    VkImage
    VkImageLayout
    VkImage
    VkImageLayout
    uint32_t
    const VkImageCopy*

```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the current layout of the source image subresource.
- `dstImage` is the destination image.
- `dstImageLayout` is the current layout of the destination image subresource.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkImageCopy` structures specifying the regions to copy.

Each region in `pRegions` is copied from the source image to the same region of the destination image. `srcImage` and `dstImage` **can** be the same image or alias the same memory.

The formats of `srcImage` and `dstImage` **must** be compatible. Formats are compatible if they share the same class, as shown in the [Compatible Formats](#) table. Depth/stencil formats **must** match exactly.

If the format of `srcImage` or `dstImage` is a [multi-planar image format](#), regions of each plane to be copied **must** be specified separately using the `srcSubresource` and `dstSubresource` members of the `VkImageCopy` structure. In this case, the `aspectMask` of the `srcSubresource` or `dstSubresource` that refers to the multi-planar image **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`. For the purposes of `vkCmdCopyImage`, each plane of a multi-planar image is treated as having the format listed in [Compatible formats of planes of multi-planar formats](#) for the plane identified by the `aspectMask` of the corresponding subresource. This applies both to `VkFormat` and to coordinates used in the copy, which correspond to texels in the *plane* rather than how these texels map to coordinates in the image as a whole.

#### Note



For example, the `VK_IMAGE_ASPECT_PLANE_1_BIT` plane of a `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM` image is compatible with an image of format `VK_FORMAT_R8G8_UNORM` and (less usefully) with the `VK_IMAGE_ASPECT_PLANE_0_BIT` plane of an image of format `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16`, as each texel is 2 bytes in size.

`vkCmdCopyImage` allows copying between *size-compatible* compressed and uncompressed internal formats. Formats are size-compatible if the texel block size of the uncompressed format is equal to the texel block size of the compressed format. Such a copy does not perform on-the-fly compression or decompression. When copying from an uncompressed format to a compressed format, each

texel of uncompressed data of the source image is copied as a raw value to the corresponding compressed texel block of the destination image. When copying from a compressed format to an uncompressed format, each compressed texel block of the source image is copied as a raw value to the corresponding texel of uncompressed data in the destination image. Thus, for example, it is legal to copy between a 128-bit uncompressed format and a compressed format which has a 128-bit sized compressed texel block representing  $4 \times 4$  texels (using 8 bits per texel), or between a 64-bit uncompressed format and a compressed format which has a 64-bit sized compressed texel block representing  $4 \times 4$  texels (using 4 bits per texel).

When copying between compressed and uncompressed formats the `extent` members represent the texel dimensions of the source image and not the destination. When copying from a compressed image to an uncompressed image the image texel dimensions written to the uncompressed image will be source extent divided by the compressed texel block dimensions. When copying from an uncompressed image to a compressed image the image texel dimensions written to the compressed image will be the source extent multiplied by the compressed texel block dimensions. In both cases the number of bytes read and the number of bytes written will be identical.

Copying to or from block-compressed images is typically done in multiples of the compressed texel block size. For this reason the `extent` **must** be a multiple of the compressed texel block dimension. There is one exception to this rule which is **required** to handle compressed images created with dimensions that are not a multiple of the compressed texel block dimensions: if the `srcImage` is compressed, then:

- If `extent.width` is not a multiple of the compressed texel block width, then (`extent.width + srcOffset.x`) **must** equal the image subresource width.
- If `extent.height` is not a multiple of the compressed texel block height, then (`extent.height + srcOffset.y`) **must** equal the image subresource height.
- If `extent.depth` is not a multiple of the compressed texel block depth, then (`extent.depth + srcOffset.z`) **must** equal the image subresource depth.

Similarly, if the `dstImage` is compressed, then:

- If `extent.width` is not a multiple of the compressed texel block width, then (`extent.width + dstOffset.x`) **must** equal the image subresource width.
- If `extent.height` is not a multiple of the compressed texel block height, then (`extent.height + dstOffset.y`) **must** equal the image subresource height.
- If `extent.depth` is not a multiple of the compressed texel block depth, then (`extent.depth + dstOffset.z`) **must** equal the image subresource depth.

This allows the last compressed texel block of the image in each non-multiple dimension to be included as a source or destination of the copy.

“[\\_422](#)” image formats that are not *multi-planar* are treated as having a  $2 \times 1$  compressed texel block for the purposes of these rules.

`vkCmdCopyImage` **can** be used to copy image data between multisample images, but both images **must** have the same number of samples.

## Valid Usage

- The source region specified by each element of `pRegions` **must** be a region that is contained within `srcImage` if the `srcImage`'s `VkFormat` is not a `multi-planar format`, and **must** be a region that is contained within the plane being copied if the `srcImage`'s `VkFormat` is a multi-planar format
- The destination region specified by each element of `pRegions` **must** be a region that is contained within `dstImage` if the `dstImage`'s `VkFormat` is not a `multi-planar format`, and **must** be a region that is contained within the plane being copied to if the `dstImage`'s `VkFormat` is a multi-planar format
- The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory
- The `format features` of `srcImage` **must** contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`.
- `srcImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag
- If `srcImage` is non-sparse then the image or *disjoint* plane to be copied **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `srcImageLayout` **must** specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `srcImageLayout` **must** be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
- The `format features` of `dstImage` **must** contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.
- `dstImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstImage` is non-sparse then the image or *disjoint* plane that is the destination of the copy **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstImageLayout` **must** specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `dstImageLayout` **must** be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
- If the `VkFormat` of each of `srcImage` and `dstImage` is not a `multi-planar format`, the `VkFormat` of each of `srcImage` and `dstImage` **must** be compatible, as defined [above](#)
- In a copy to or from a plane of a `multi-planar image`, the `VkFormat` of the image and plane **must** be compatible according to the description of `compatible planes` for the plane being copied
- When a copy is performed to or from an image with a `multi-planar format`, the `aspectMask` of the `srcSubresource` and/or `dstSubresource` that refers to the multi-planar image **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT` (with `VK_IMAGE_ASPECT_PLANE_2_BIT` valid only for a `VkFormat` with three planes)
- The sample count of `srcImage` and `dstImage` **must** match
- If `commandBuffer` is an unprotected command buffer, then `srcImage` **must** not be a protected image

- If `commandBuffer` is an unprotected command buffer, then `dstImage` **must** not be a protected image
- If `commandBuffer` is a protected command buffer, then `dstImage` **must** not be an unprotected image
- The `srcSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created
- The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created
- The `srcSubresource.baseArrayLayer` + `srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created
- The `dstSubresource.baseArrayLayer` + `dstSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created
- The `srcOffset` and `extent` members of each element of `pRegions` **must** respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`
- The `dstOffset` and `extent` members of each element of `pRegions` **must** respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`
- `dstImage` and `srcImage` **must** not have been created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `srcImage` **must** be a valid `VkImage` handle
- `srcImageLayout` **must** be a valid `VkImageLayout` value
- `dstImage` **must** be a valid `VkImage` handle
- `dstImageLayout` **must** be a valid `VkImageLayout` value
- `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkImageCopy` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- `regionCount` **must** be greater than `0`
- Each of `commandBuffer`, `dstImage`, and `srcImage` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Transfer Graphics Compute	Transfer

The `VkImageCopy` structure is defined as:

```
typedef struct VkImageCopy {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D             srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D             dstOffset;
    VkExtent3D              extent;
} VkImageCopy;
```

- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively.
- `srcOffset` and `dstOffset` select the initial `x`, `y`, and `z` offsets in texels of the sub-regions of the source and destination image data.
- `extent` is the size in texels of the image to copy in `width`, `height` and `depth`.

For `VK_IMAGE_TYPE_3D` images, copies are performed slice by slice starting with the `z` member of the `srcOffset` or `dstOffset`, and copying `depth` slices. For images with multiple layers, copies are performed layer by layer starting with the `baseArrayLayer` member of the `srcSubresource` or `dstSubresource` and copying `layerCount` layers. Image data **can** be copied between images with different image types. If one image is `VK_IMAGE_TYPE_3D` and the other image is `VK_IMAGE_TYPE_2D` with multiple layers, then each slice is copied to or from a different layer.

Copies involving a [multi-planar image format](#) specify the region to be copied in terms of the *plane* to be copied, not the coordinates of the multi-planar image. This means that copies accessing the R/B planes of “`_422`” format images **must** fit the copied region within half the `width` of the parent image, and that copies accessing the R/B planes of “`_420`” format images **must** fit the copied region within half the `width` and `height` of the parent image.

## Valid Usage

- If neither the calling command's `srcImage` nor the calling command's `dstImage` has a [multi-planar image format](#) then the `aspectMask` member of `srcSubresource` and `dstSubresource` **must** match
- If the calling command's `srcImage` has a `VkFormat` with [two planes](#) then the `srcSubresource aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`
- If the calling command's `srcImage` has a `VkFormat` with [three planes](#) then the `srcSubresource aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`
- If the calling command's `dstImage` has a `VkFormat` with [two planes](#) then the `dstSubresource aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`
- If the calling command's `dstImage` has a `VkFormat` with [three planes](#) then the `dstSubresource aspectMask` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`
- If the calling command's `srcImage` has a [multi-planar image format](#) and the `dstImage` does not have a multi-planar image format, the `dstSubresource aspectMask` **must** be `VK_IMAGE_ASPECT_COLOR_BIT`
- If the calling command's `dstImage` has a [multi-planar image format](#) and the `srcImage` does not have a multi-planar image format, the `srcSubresource aspectMask` **must** be `VK_IMAGE_ASPECT_COLOR_BIT`
- The number of slices of the `extent` (for 3D) or layers of the `srcSubresource` (for non-3D) **must** match the number of slices of the `extent` (for 3D) or layers of the `dstSubresource` (for non-3D)
- If either of the calling command's `srcImage` or `dstImage` parameters are of `VkImageType VK_IMAGE_TYPE_3D`, the `baseArrayLayer` and `layerCount` members of the corresponding subresource **must** be `0` and `1`, respectively
- The `aspectMask` member of `srcSubresource` **must** specify aspects present in the calling command's `srcImage`
- The `aspectMask` member of `dstSubresource` **must** specify aspects present in the calling command's `dstImage`
- `srcOffset.x` and  $(\text{extent.width} + \text{srcOffset.x})$  **must** both be greater than or equal to `0` and less than or equal to the source image subresource width
- `srcOffset.y` and  $(\text{extent.height} + \text{srcOffset.y})$  **must** both be greater than or equal to `0` and less than or equal to the source image subresource height
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_1D`, then `srcOffset.y` **must** be `0` and `extent.height` **must** be `1`.
- `srcOffset.z` and  $(\text{extent.depth} + \text{srcOffset.z})$  **must** both be greater than or equal to `0` and less than or equal to the source image subresource depth
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_1D`, then `srcOffset.z` **must** be `0`

and `extent.depth` must be 1.

- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_1D`, then `dstOffset.z` must be 0 and `extent.depth` must be 1.
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_2D`, then `srcOffset.z` must be 0.
- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_2D`, then `dstOffset.z` must be 0.
- If both `srcImage` and `dstImage` are of type `VK_IMAGE_TYPE_2D` then `extent.depth` must be 1.
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_2D`, and the `dstImage` is of type `VK_IMAGE_TYPE_3D`, then `extent.depth` must equal to the `layerCount` member of `srcSubresource`.
- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_2D`, and the `srcImage` is of type `VK_IMAGE_TYPE_3D`, then `extent.depth` must equal to the `layerCount` member of `dstSubresource`.
- `dstOffset.x` and  $(\text{extent.width} + \text{dstOffset.x})$  must both be greater than or equal to 0 and less than or equal to the destination image subresource width
- `dstOffset.y` and  $(\text{extent.height} + \text{dstOffset.y})$  must both be greater than or equal to 0 and less than or equal to the destination image subresource height
- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_1D`, then `dstOffset.y` must be 0 and `extent.height` must be 1.
- `dstOffset.z` and  $(\text{extent.depth} + \text{dstOffset.z})$  must both be greater than or equal to 0 and less than or equal to the destination image subresource depth
- If the calling command's `srcImage` is a compressed image, or a *single-plane*, “[\\_422](#)” image format, all members of `srcOffset` must be a multiple of the corresponding dimensions of the compressed texel block
- If the calling command's `srcImage` is a compressed image, or a *single-plane*, “[\\_422](#)” image format, `extent.width` must be a multiple of the compressed texel block width or  $(\text{extent.width} + \text{srcOffset.x})$  must equal the source image subresource width
- If the calling command's `srcImage` is a compressed image, or a *single-plane*, “[\\_422](#)” image format, `extent.height` must be a multiple of the compressed texel block height or  $(\text{extent.height} + \text{srcOffset.y})$  must equal the source image subresource height
- If the calling command's `srcImage` is a compressed image, or a *single-plane*, “[\\_422](#)” image format, `extent.depth` must be a multiple of the compressed texel block depth or  $(\text{extent.depth} + \text{srcOffset.z})$  must equal the source image subresource depth
- If the calling command's `dstImage` is a compressed format image, or a *single-plane*, “[\\_422](#)” image format, all members of `dstOffset` must be a multiple of the corresponding dimensions of the compressed texel block
- If the calling command's `dstImage` is a compressed format image, or a *single-plane*, “[\\_422](#)” image format, `extent.width` must be a multiple of the compressed texel block width or  $(\text{extent.width} + \text{dstOffset.x})$  must equal the destination image subresource width
- If the calling command's `dstImage` is a compressed format image, or a *single-plane*, “[\\_422](#)” image format, `extent.height` must be a multiple of the compressed texel block height or  $(\text{extent.height} + \text{dstOffset.y})$  must equal the destination image subresource height

- If the calling command's `dstImage` is a compressed format image, or a *single-plane*, “[\\_422](#)” image format, `extent.depth` **must** be a multiple of the compressed texel block depth or `(extent.depth + dstOffset.z)` **must** equal the destination image subresource depth

## Valid Usage (Implicit)

- `srcSubresource` **must** be a valid `VkImageSubresourceLayers` structure
- `dstSubresource` **must** be a valid `VkImageSubresourceLayers` structure

The `VkImageSubresourceLayers` structure is defined as:

```
typedef struct VkImageSubresourceLayers {
    VkImageAspectFlags aspectMask;
    uint32_t          mipLevel;
    uint32_t          baseArrayLayer;
    uint32_t          layerCount;
} VkImageSubresourceLayers;
```

- `aspectMask` is a combination of `VkImageAspectFlagBits`, selecting the color, depth and/or stencil aspects to be copied.
- `mipLevel` is the mipmap level to copy from.
- `baseArrayLayer` and `layerCount` are the starting layer and number of layers to copy.

## Valid Usage

- If `aspectMask` contains `VK_IMAGE_ASPECT_COLOR_BIT`, it **must** not contain either of `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`
- `aspectMask` **must** not contain `VK_IMAGE_ASPECT_METADATA_BIT`
- `aspectMask` **must** not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`.
- `layerCount` **must** be greater than 0

## Valid Usage (Implicit)

- `aspectMask` **must** be a valid combination of `VkImageAspectFlagBits` values
- `aspectMask` **must** not be 0

## 19.4. Copying Data Between Buffers and Images

To copy data from a buffer object to an image object, call:

```
void vkCmdCopyBufferToImage(  
    VkCommandBuffer  
    VkBuffer  
    VkImage  
    VkImageLayout  
    uint32_t  
    const VkBufferImageCopy*
```

```
        commandBuffer,  
        srcBuffer,  
        dstImage,  
        dstImageLayout,  
        regionCount,  
        pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcBuffer` is the source buffer.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the copy.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy` structures specifying the regions to copy.

Each region in `pRegions` is copied from the specified region of the source buffer to the specified region of the destination image.

If the format of `dstImage` is a [multi-planar image format](#)), regions of each plane to be a target of a copy **must** be specified separately using the `pRegions` member of the `VkBufferImageCopy` structure. In this case, the `aspectMask` of `imageSubresource` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`. For the purposes of `vkCmdCopyBufferToImage`, each plane of a multi-planar image is treated as having the format listed in [Compatible formats of planes of multi-planar formats](#) for the plane identified by the `aspectMask` of the corresponding subresource. This applies both to `VkFormat` and to coordinates used in the copy, which correspond to texels in the *plane* rather than how these texels map to coordinates in the image as a whole.

## Valid Usage

- `srcBuffer` **must** be large enough to contain all buffer locations that are accessed according to [Buffer and Image Addressing](#), for each element of `pRegions`
- The image region specified by each element of `pRegions` **must** be a region that is contained within `dstImage` if the `dstImage`'s `VkFormat` is not a [multi-planar format](#), and **must** be a region that is contained within the plane being copied to if the `dstImage`'s `VkFormat` is a multi-planar format
- The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory
- `srcBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag
- The [format features](#) of `dstImage` **must** contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.
- If `srcBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstImage` **must** have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`
- `dstImageLayout` **must** specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `dstImageLayout` **must** be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
- If `commandBuffer` is an unprotected command buffer, then `srcBuffer` **must** not be a protected buffer
- If `commandBuffer` is an unprotected command buffer, then `dstImage` **must** not be a protected image
- If `commandBuffer` is a protected command buffer, then `dstImage` **must** not be an unprotected image
- The `imageSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created
- The `imageSubresource.baseArrayLayer + imageSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created
- The `imageOffset` and `imageExtent` members of each element of `pRegions` **must** respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in [VkQueueFamilyProperties](#)
- `dstImage` **must** not have been created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `srcBuffer` **must** be a valid `VkBuffer` handle
- `dstImage` **must** be a valid `VkImage` handle
- `dstImageLayout` **must** be a valid `VkImageLayout` value
- `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- `regionCount` **must** be greater than `0`
- Each of `commandBuffer`, `dstImage`, and `srcBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Transfer Graphics Compute	Transfer

To copy data from an image object to a buffer object, call:

```
void vkCmdCopyImageToBuffer(  
    VkCommandBuffer  
    VkImage  
    VkImageLayout  
    VkBuffer  
    uint32_t  
    const VkBufferImageCopy*  
        commandBuffer,  
        srcImage,  
        srcImageLayout,  
        dstBuffer,  
        regionCount,  
        pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the copy.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy` structures specifying the regions to copy.

Each region in `pRegions` is copied from the specified region of the source image to the specified region of the destination buffer.

If the `VkFormat` of `srcImage` is a [multi-planar image format](#), regions of each plane to be a source of a copy **must** be specified separately using the `pRegions` member of the `VkBufferImageCopy` structure. In this case, the `aspectMask` of `imageSubresource` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`. For the purposes of `vkCmdCopyBufferToImage`, each plane of a multi-planar image is treated as having the format listed in [Compatible formats of planes of multi-planar formats](#) for the plane identified by the `aspectMask` of the corresponding subresource. This applies both to `VkFormat` and to coordinates used in the copy, which correspond to texels in the *plane* rather than how these texels map to coordinates in the image as a whole.

## Valid Usage

- The image region specified by each element of `pRegions` **must** be a region that is contained within `srcImage` if the `srcImage`'s `VkFormat` is not a [multi-planar format](#), and **must** be a region that is contained within the plane being copied if the `srcImage`'s `VkFormat` is a multi-planar format
- `dstBuffer` **must** be large enough to contain all buffer locations that are accessed according to [Buffer and Image Addressing](#), for each element of `pRegions`
- The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory
- The [format features](#) of `srcImage` **must** contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`.
- `srcImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag
- If `srcImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `srcImage` **must** have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`
- `srcImageLayout` **must** specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `srcImageLayout` **must** be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- If `commandBuffer` is an unprotected command buffer, then `srcImage` **must** not be a protected image
- If `commandBuffer` is an unprotected command buffer, then `dstBuffer` **must** not be a protected buffer
- If `commandBuffer` is a protected command buffer, then `dstBuffer` **must** not be an unprotected buffer
- The `imageSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created
- The `imageSubresource.baseArrayLayer + imageSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created
- The `imageOffset` and `imageExtent` members of each element of `pRegions` **must** respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`
- `srcImage` **must** not have been created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `srcImage` **must** be a valid `VkImage` handle
- `srcImageLayout` **must** be a valid `VkImageLayout` value
- `dstBuffer` **must** be a valid `VkBuffer` handle
- `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- This command **must** only be called outside of a render pass instance
- `regionCount` **must** be greater than `0`
- Each of `commandBuffer`, `dstBuffer`, and `srcImage` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Transfer Graphics Compute	Transfer

For both `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`, each element of `pRegions` is a structure defined as:

```

typedef struct VkBufferImageCopy {
    VkDeviceSize          bufferOffset;
    uint32_t              bufferRowLength;
    uint32_t              bufferImageHeight;
    VkImageSubresourceLayers imageSubresource;
    VkOffset3D            imageOffset;
    VkExtent3D            imageExtent;
} VkBufferImageCopy;

```

- **bufferOffset** is the offset in bytes from the start of the buffer object where the image data is copied from or to.
- **bufferRowLength** and **bufferImageHeight** specify in texels a subregion of a larger two- or three-dimensional image in buffer memory, and control the addressing calculations. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the **imageExtent**.
- **imageSubresource** is a [VkImageSubresourceLayers](#) used to specify the specific image subresources of the image used for the source or destination image data.
- **imageOffset** selects the initial **x**, **y**, **z** offsets in texels of the sub-region of the source or destination image data.
- **imageExtent** is the size in texels of the image to copy in **width**, **height** and **depth**.

When copying to or from a depth or stencil aspect, the data in buffer memory uses a layout that is a (mostly) tightly packed representation of the depth or stencil data. Specifically:

- data copied to or from the stencil aspect of any depth/stencil format is tightly packed with one **VK\_FORMAT\_S8\_UINT** value per texel.
- data copied to or from the depth aspect of a **VK\_FORMAT\_D16\_UNORM** or **VK\_FORMAT\_D16\_UNORM\_S8\_UINT** format is tightly packed with one **VK\_FORMAT\_D16\_UNORM** value per texel.
- data copied to or from the depth aspect of a **VK\_FORMAT\_D32\_SFLOAT** or **VK\_FORMAT\_D32\_SFLOAT\_S8\_UINT** format is tightly packed with one **VK\_FORMAT\_D32\_SFLOAT** value per texel.
- data copied to or from the depth aspect of a **VK\_FORMAT\_X8\_D24\_UNORM\_PACK32** or **VK\_FORMAT\_D24\_UNORM\_S8\_UINT** format is packed with one 32-bit word per texel with the D24 value in the LSBs of the word, and undefined values in the eight MSBs.

#### *Note*



To copy both the depth and stencil aspects of a depth/stencil format, two entries in **pRegions** **can** be used, where one specifies the depth aspect in **imageSubresource**, and the other specifies the stencil aspect.

Because depth or stencil aspect buffer to image copies **may** require format conversions on some implementations, they are not supported on queues that do not support graphics.

When copying to a depth aspect, and the **VK\_EXT\_depth\_range\_unrestricted** extension is not enabled, the data in buffer memory **must** be in the range [0,1], or the resulting values are

undefined.

Copies are done layer by layer starting with image layer `baseArrayLayer` member of `imageSubresource`. `layerCount` layers are copied from the source image or to the destination image.

## Valid Usage

- If the calling command's `VkImage` parameter's format is not a depth/stencil format or a [multi-planar format](#), then `bufferOffset` **must** be a multiple of the format's texel block size.
- If the calling command's `VkImage` parameter's format is a [multi-planar format](#), then `bufferOffset` **must** be a multiple of the element size of the compatible format for the format and the `aspectMask` of the `imageSubresource` as defined in [Compatible formats of planes of multi-planar formats](#)
- `bufferOffset` **must** be a multiple of 4
- `bufferRowLength` **must** be 0, or greater than or equal to the `width` member of `imageExtent`
- `bufferImageHeight` **must** be 0, or greater than or equal to the `height` member of `imageExtent`
- `imageOffset.x` and (`imageExtent.width + imageOffset.x`) **must** both be greater than or equal to 0 and less than or equal to the image subresource width where this refers to the width of the *plane* of the image involved in the copy in the case of a [multi-planar format](#)
- `imageOffset.y` and (`imageExtent.height + imageOffset.y`) **must** both be greater than or equal to 0 and less than or equal to the image subresource height where this refers to the height of the *plane* of the image involved in the copy in the case of a [multi-planar format](#)
- If the calling command's `srcImage` (`vkCmdCopyImageToBuffer`) or `dstImage` (`vkCmdCopyBufferToImage`) is of type `VK_IMAGE_TYPE_1D`, then `imageOffset.y` **must** be 0 and `imageExtent.height` **must** be 1.
- `imageOffset.z` and (`imageExtent.depth + imageOffset.z`) **must** both be greater than or equal to 0 and less than or equal to the image subresource depth
- If the calling command's `srcImage` (`vkCmdCopyImageToBuffer`) or `dstImage` (`vkCmdCopyBufferToImage`) is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then `imageOffset.z` **must** be 0 and `imageExtent.depth` **must** be 1
- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane, “\_422”* image format, `bufferRowLength` **must** be a multiple of the compressed texel block width
- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane, “\_422”* image format, `bufferImageHeight` **must** be a multiple of the compressed texel block height
- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane, “\_422”* image format, all members of `imageOffset` **must** be a multiple of the corresponding dimensions of the compressed texel block
- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane, “\_422”* image format, `bufferOffset` **must** be a multiple of the compressed texel block size in bytes
- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane, “\_422”* image format, `imageExtent.width` **must** be a multiple of the compressed texel block width or (`imageExtent.width + imageOffset.x`) **must** equal the image subresource width
- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane, “\_422”* image format, `imageExtent.height` **must** be a multiple of the compressed texel block height or (`imageExtent.height + imageOffset.y`) **must** equal the image subresource height
- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane, “\_422”* image format, `imageExtent.depth` **must** be a multiple of the compressed texel block depth or (`imageExtent.depth + imageOffset.z`) **must** equal the image subresource depth

`_422`" image format, `imageExtent.height` **must** be a multiple of the compressed texel block height or (`imageExtent.height + imageOffset.y`) **must** equal the image subresource height

- If the calling command's `VkImage` parameter is a compressed image, or a *single-plane*, “`_422`” image format, `imageExtent.depth` **must** be a multiple of the compressed texel block depth or (`imageExtent.depth + imageOffset.z`) **must** equal the image subresource depth
- The `aspectMask` member of `imageSubresource` **must** specify aspects present in the calling command's `VkImage` parameter
- If the calling command's `VkImage` parameter's format is a *multi-planar format*, then the `aspectMask` member of `imageSubresource` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT` (with `VK_IMAGE_ASPECT_PLANE_2_BIT` valid only for image formats with three planes)
- The `aspectMask` member of `imageSubresource` **must** only have a single bit set
- If the calling command's `VkImage` parameter is of `VkImageType VK_IMAGE_TYPE_3D`, the `baseArrayLayer` and `layerCount` members of `imageSubresource` **must** be `0` and `1`, respectively

### Valid Usage (Implicit)

- `imageSubresource` **must** be a valid `VkImageSubresourceLayers` structure

#### 19.4.1. Buffer and Image Addressing

Pseudocode for image/buffer addressing of uncompressed formats is:

```
rowLength = region->bufferRowLength;  
if (rowLength == 0)  
    rowLength = region->imageExtent.width;  
  
imageHeight = region->bufferImageHeight;  
if (imageHeight == 0)  
    imageHeight = region->imageExtent.height;  
  
texelBlockSize = <texel block size of the format of the src/dstImage>;  
  
address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x)  
* texelBlockSize;  
  
where x,y,z range from (0,0,0) to region->imageExtent.{width,height,depth}.
```

Note that `imageOffset` does not affect addressing calculations for buffer memory. Instead, `bufferOffset` can be used to select the starting address in buffer memory.

For block-compressed formats, all parameters are still specified in texels rather than compressed texel blocks, but the addressing math operates on whole compressed texel blocks. Pseudocode for compressed copy addressing is:

```

rowLength = region->bufferRowLength;
if (rowLength == 0)
    rowLength = region->imageExtent.width;

imageHeight = region->bufferImageHeight;
if (imageHeight == 0)
    imageHeight = region->imageExtent.height;

compressedTexelBlockSizeInBytes = <compressed texel block size taken from the src
/dstImage>;
rowLength /= compressedTexelBlockWidth;
imageHeight /= compressedTexelBlockHeight;

address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x)
* compressedTexelBlockSizeInBytes;

where x,y,z range from (0,0,0) to region->imageExtent.{width/
compressedTexelBlockWidth,height/compressedTexelBlockHeight,depth/compressedTexelBlock
Depth}.

```

Copying to or from block-compressed images is typically done in multiples of the compressed texel block size. For this reason the `imageExtent` **must** be a multiple of the compressed texel block dimension. There is one exception to this rule which is **required** to handle compressed images created with dimensions that are not a multiple of the compressed texel block dimensions:

- If `imageExtent.width` is not a multiple of the compressed texel block width, then `(imageExtent.width + imageOffset.x)` **must** equal the image subresource width.
- If `imageExtent.height` is not a multiple of the compressed texel block height, then `(imageExtent.height + imageOffset.y)` **must** equal the image subresource height.
- If `imageExtent.depth` is not a multiple of the compressed texel block depth, then `(imageExtent.depth + imageOffset.z)` **must** equal the image subresource depth.

This allows the last compressed texel block of the image in each non-multiple dimension to be included as a source or destination of the copy.

## 19.5. Image Copies with Scaling

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:

```

void vkCmdBlitImage(
    VkCommandBuffer
    VkImage
    VkImageLayout
    VkImage
    VkImageLayout
    uint32_t
    const VkImageBlit*
    VkFilter
        commandBuffer,
        srcImage,
        srcImageLayout,
        dstImage,
        dstImageLayout,
        regionCount,
        pRegions,
        filter);

```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the blit.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the blit.
- `regionCount` is the number of regions to blit.
- `pRegions` is a pointer to an array of `VkImageBlit` structures specifying the regions to blit.
- `filter` is a `VkFilter` specifying the filter to apply if the blits require scaling.

`vkCmdBlitImage` must not be used for multisampled source or destination images. Use `vkCmdResolveImage` for this purpose.

As the sizes of the source and destination extents **can** differ in any dimension, texels in the source extent are scaled and filtered to the destination extent. Scaling occurs via the following operations:

- For each destination texel, the integer coordinate of that texel is converted to an unnormalized texture coordinate, using the effective inverse of the equations described in [unnormalized to integer conversion](#):

$$u_{\text{base}} = i + \frac{1}{2}$$

$$v_{\text{base}} = j + \frac{1}{2}$$

$$w_{\text{base}} = k + \frac{1}{2}$$

- These base coordinates are then offset by the first destination offset:

$$u_{\text{offset}} = u_{\text{base}} - x_{\text{dst0}}$$

$$v_{\text{offset}} = v_{\text{base}} - y_{\text{dst0}}$$

$$w_{\text{offset}} = w_{\text{base}} - z_{\text{dst0}}$$

$$a_{\text{offset}} = a - \text{baseArrayCount}_{\text{dst}}$$

- The scale is determined from the source and destination regions, and applied to the offset coordinates:

$$\text{scale\_u} = (\mathbf{x}_{\text{src1}} - \mathbf{x}_{\text{src0}}) / (\mathbf{x}_{\text{dst1}} - \mathbf{x}_{\text{dst0}})$$

$$\text{scale\_v} = (\mathbf{y}_{\text{src1}} - \mathbf{y}_{\text{src0}}) / (\mathbf{y}_{\text{dst1}} - \mathbf{y}_{\text{dst0}})$$

$$\text{scale\_w} = (\mathbf{z}_{\text{src1}} - \mathbf{z}_{\text{src0}}) / (\mathbf{z}_{\text{dst1}} - \mathbf{z}_{\text{dst0}})$$

$$\mathbf{u}_{\text{scaled}} = \mathbf{u}_{\text{offset}} * \text{scale}_u$$

$$\mathbf{v}_{\text{scaled}} = \mathbf{v}_{\text{offset}} * \text{scale}_v$$

$$\mathbf{w}_{\text{scaled}} = \mathbf{w}_{\text{offset}} * \text{scale}_w$$

- Finally the source offset is added to the scaled coordinates, to determine the final unnormalized coordinates used to sample from `srcImage`:

$$\mathbf{u} = \mathbf{u}_{\text{scaled}} + \mathbf{x}_{\text{src0}}$$

$$\mathbf{v} = \mathbf{v}_{\text{scaled}} + \mathbf{y}_{\text{src0}}$$

$$\mathbf{w} = \mathbf{w}_{\text{scaled}} + \mathbf{z}_{\text{src0}}$$

$$q = \text{mipLevel}$$

$$a = a_{\text{offset}} + \text{baseArrayCount}_{\text{src}}$$

These coordinates are used to sample from the source image, as described in [Image Operations chapter](#), with the filter mode equal to that of `filter`, a mipmap mode of `VK_SAMPLER_MIPMAP_MODE_NEAREST` and an address mode of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`. Implementations **must** clamp at the edge of the source image, and **may** additionally clamp to the edge of the source region.

*Note*



Due to allowable rounding errors in the generation of the source texture coordinates, it is not always possible to guarantee exactly which source texels will be sampled for a given blit. As rounding errors are implementation dependent, the exact results of a blitting operation are also implementation dependent.

Blits are done layer by layer starting with the `baseArrayLayer` member of `srcSubresource` for the source and `dstSubresource` for the destination. `layerCount` layers are blitted to the destination image.

3D textures are blitted slice by slice. Slices in the source region bounded by `srcOffsets[0].z` and `srcOffsets[1].z` are copied to slices in the destination region bounded by `dstOffsets[0].z` and `dstOffsets[1].z`. For each destination slice, a source z coordinate is linearly interpolated between `srcOffsets[0].z` and `srcOffsets[1].z`. If the `filter` parameter is `VK_FILTER_LINEAR` then the value

sampled from the source image is taken by doing linear filtering using the interpolated  $z$  coordinate. If `filter` parameter is `VK_FILTER_NEAREST` then the value sampled from the source image is taken from the single nearest slice, with an implementation-dependent arithmetic rounding mode.

The following filtering and conversion rules apply:

- Integer formats **can** only be converted to other integer formats with the same signedness.
- No format conversion is supported between depth/stencil images. The formats **must** match.
- Format conversions on unorm, snorm, unscaled and packed float formats of the copied aspect of the image are performed by first converting the pixels to float values.
- For sRGB source formats, nonlinear RGB values are converted to linear representation prior to filtering.
- After filtering, the float values are first clamped and then cast to the destination image format. In case of sRGB destination format, linear RGB values are converted to nonlinear representation before writing the pixel to the image.

Signed and unsigned integers are converted by first clamping to the representable range of the destination format, then casting the value.

## Valid Usage

- The source region specified by each element of `pRegions` **must** be a region that is contained within `srcImage`
- The destination region specified by each element of `pRegions` **must** be a region that is contained within `dstImage`
- The union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory with any texel that **may** be sampled during the blit operation
- The `format features` of `srcImage` **must** contain `VK_FORMAT_FEATURE_BLIT_SRC_BIT`.
- `srcImage` **must** not use a format listed in `Formats requiring sampler Y'CBCR conversion for VK_IMAGE_ASPECT_COLOR_BIT image views`
- `srcImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag
- If `srcImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `srcImageLayout` **must** specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `srcImageLayout` **must** be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- The `format features` of `dstImage` **must** contain `VK_FORMAT_FEATURE_BLIT_DST_BIT`.
- `dstImage` **must** not use a format listed in `Formats requiring sampler Y'CBCR conversion for VK_IMAGE_ASPECT_COLOR_BIT image views`
- `dstImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstImageLayout` **must** specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `dstImageLayout` **must** be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- The sample count of `srcImage` and `dstImage` **must** both be equal to `VK_SAMPLE_COUNT_1_BIT`
- If either of `srcImage` or `dstImage` was created with a signed integer `VkFormat`, the other **must** also have been created with a signed integer `VkFormat`
- If either of `srcImage` or `dstImage` was created with an unsigned integer `VkFormat`, the other **must** also have been created with an unsigned integer `VkFormat`
- If either of `srcImage` or `dstImage` was created with a depth/stencil format, the other **must** have exactly the same format
- If `srcImage` was created with a depth/stencil format, `filter` **must** be `VK_FILTER_NEAREST`
- `srcImage` **must** have been created with a `samples` value of `VK_SAMPLE_COUNT_1_BIT`
- `dstImage` **must** have been created with a `samples` value of `VK_SAMPLE_COUNT_1_BIT`
- If `filter` is `VK_FILTER_LINEAR`, then the `format features` of `srcImage` **must** contain

`VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- If `filter` is `VK_FILTER_CUBIC_EXT`, then the `format` features of `srcImage` **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.
- If `filter` is `VK_FILTER_CUBIC_EXT`, `srcImage` **must** have a `VkImageType` of `VK_IMAGE_TYPE_2D`
- If `commandBuffer` is an unprotected command buffer, then `srcImage` **must** not be a protected image
- If `commandBuffer` is an unprotected command buffer, then `dstImage` **must** not be a protected image
- If `commandBuffer` is a protected command buffer, then `dstImage` **must** not be an unprotected image
- The `srcSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created
- The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created
- The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created
- The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created
- `dstImage` and `srcImage` **must** not have been created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `srcImage` **must** be a valid `VkImage` handle
- `srcImageLayout` **must** be a valid `VkImageLayout` value
- `dstImage` **must** be a valid `VkImage` handle
- `dstImageLayout` **must** be a valid `VkImageLayout` value
- `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkImageBlit` structures
- `filter` **must** be a valid `VkFilter` value
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called outside of a render pass instance
- `regionCount` **must** be greater than 0
- Each of `commandBuffer`, `dstImage`, and `srcImage` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics	Transfer

The `VkImageBlit` structure is defined as:

```
typedef struct VkImageBlit {  
    VkImageSubresourceLayers    srcSubresource;  
    VkOffset3D                 srcOffsets[2];  
    VkImageSubresourceLayers    dstSubresource;  
    VkOffset3D                 dstOffsets[2];  
} VkImageBlit;
```

- `srcSubresource` is the subresource to blit from.
- `srcOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the source region within `srcSubresource`.
- `dstSubresource` is the subresource to blit into.
- `dstOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the destination region within `dstSubresource`.

For each element of the `pRegions` array, a blit operation is performed the specified source and destination regions.

## Valid Usage

- The `aspectMask` member of `srcSubresource` and `dstSubresource` **must** match
- The `layerCount` member of `srcSubresource` and `dstSubresource` **must** match
- If either of the calling command's `srcImage` or `dstImage` parameters are of `VkImageType VK_IMAGE_TYPE_3D`, the `baseArrayLayer` and `layerCount` members of both `srcSubresource` and `dstSubresource` **must** be `0` and `1`, respectively
- The `aspectMask` member of `srcSubresource` **must** specify aspects present in the calling command's `srcImage`
- The `aspectMask` member of `dstSubresource` **must** specify aspects present in the calling command's `dstImage`
- `srcOffset[0].x` and `srcOffset[1].x` **must** both be greater than or equal to `0` and less than or equal to the source image subresource width
- `srcOffset[0].y` and `srcOffset[1].y` **must** both be greater than or equal to `0` and less than or equal to the source image subresource height
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_1D`, then `srcOffset[0].y` **must** be `0` and `srcOffset[1].y` **must** be `1`.
- `srcOffset[0].z` and `srcOffset[1].z` **must** both be greater than or equal to `0` and less than or equal to the source image subresource depth
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then `srcOffset[0].z` **must** be `0` and `srcOffset[1].z` **must** be `1`.
- `dstOffset[0].x` and `dstOffset[1].x` **must** both be greater than or equal to `0` and less than or equal to the destination image subresource width
- `dstOffset[0].y` and `dstOffset[1].y` **must** both be greater than or equal to `0` and less than or equal to the destination image subresource height
- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_1D`, then `dstOffset[0].y` **must** be `0` and `dstOffset[1].y` **must** be `1`.
- `dstOffset[0].z` and `dstOffset[1].z` **must** both be greater than or equal to `0` and less than or equal to the destination image subresource depth
- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then `dstOffset[0].z` **must** be `0` and `dstOffset[1].z` **must** be `1`.

## Valid Usage (Implicit)

- `srcSubresource` **must** be a valid `VkImageSubresourceLayers` structure
- `dstSubresource` **must** be a valid `VkImageSubresourceLayers` structure

## 19.6. Resolving Multisample Images

To resolve a multisample image to a non-multisample image, call:

```
void vkCmdResolveImage(  
    VkCommandBuffer  
    VkImage  
    VkImageLayout  
    VkImage  
    VkImageLayout  
    uint32_t  
    const VkImageResolve*  
                                commandBuffer,  
                                srcImage,  
                                srcImageLayout,  
                                dstImage,  
                                dstImageLayout,  
                                regionCount,  
                                pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the resolve.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the resolve.
- `regionCount` is the number of regions to resolve.
- `pRegions` is a pointer to an array of `VkImageResolve` structures specifying the regions to resolve.

During the resolve the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination. If the source formats are floating-point or normalized types, the sample values for each pixel are resolved in an implementation-dependent manner. If the source formats are integer types, a single sample's value is selected for each pixel.

`srcOffset` and `dstOffset` select the initial `x`, `y`, and `z` offsets in texels of the sub-regions of the source and destination image data. `extent` is the size in texels of the source image to resolve in `width`, `height` and `depth`.

Resolves are done layer by layer starting with `baseArrayLayer` member of `srcSubresource` for the source and `dstSubresource` for the destination. `layerCount` layers are resolved to the destination image.

## Valid Usage

- The source region specified by each element of `pRegions` **must** be a region that is contained within `srcImage`
- The destination region specified by each element of `pRegions` **must** be a region that is contained within `dstImage`
- The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory
- If `srcImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `srcImage` **must** have a sample count equal to any valid sample count value other than `VK_SAMPLE_COUNT_1_BIT`
- If `dstImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstImage` **must** have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`
- `srcImageLayout` **must** specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `srcImageLayout` **must** be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- `dstImageLayout` **must** specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`
- `dstImageLayout` **must** be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- The `format features` of `dstImage` **must** contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.
- `srcImage` and `dstImage` **must** have been created with the same image format
- If `commandBuffer` is an unprotected command buffer, then `srcImage` **must** not be a protected image
- If `commandBuffer` is an unprotected command buffer, then `dstImage` **must** not be a protected image
- If `commandBuffer` is a protected command buffer, then `dstImage` **must** not be an unprotected image
- The `srcSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created
- The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created
- The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created
- The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo`

when `dstImage` was created

- `dstImage` and `srcImage` **must** not have been created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `srcImage` **must** be a valid `VkImage` handle
- `srcImageLayout` **must** be a valid `VkImageLayout` value
- `dstImage` **must** be a valid `VkImage` handle
- `dstImageLayout` **must** be a valid `VkImageLayout` value
- `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkImageResolve` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called outside of a render pass instance
- `regionCount` **must** be greater than `0`
- Each of `commandBuffer`, `dstImage`, and `srcImage` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Graphics	Transfer

The `VkImageResolve` structure is defined as:

```
typedef struct VkImageResolve {
    VkImageSubresourceLayers    srcSubresource;
    VkOffset3D                  srcOffset;
    VkImageSubresourceLayers    dstSubresource;
    VkOffset3D                  dstOffset;
    VkExtent3D                  extent;
} VkImageResolve;
```

- **srcSubresource** and **dstSubresource** are [VkImageSubresourceLayers](#) structures specifying the image subresources of the images used for the source and destination image data, respectively. Resolve of depth/stencil images is not supported.
- **srcOffset** and **dstOffset** select the initial **x**, **y**, and **z** offsets in texels of the sub-regions of the source and destination image data.
- **extent** is the size in texels of the source image to resolve in **width**, **height** and **depth**.

## Valid Usage

- The `aspectMask` member of `srcSubresource` and `dstSubresource` **must** only contain `VK_IMAGE_ASPECT_COLOR_BIT`
- The `layerCount` member of `srcSubresource` and `dstSubresource` **must** match
- If either of the calling command's `srcImage` or `dstImage` parameters are of `VkImageType VK_IMAGE_TYPE_3D`, the `baseArrayLayer` and `layerCount` members of both `srcSubresource` and `dstSubresource` **must** be `0` and `1`, respectively
- `srcOffset.x` and `(extent.width + srcOffset.x)` **must** both be greater than or equal to `0` and less than or equal to the source image subresource width
- `srcOffset.y` and `(extent.height + srcOffset.y)` **must** both be greater than or equal to `0` and less than or equal to the source image subresource height
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_1D`, then `srcOffset.y` **must** be `0` and `extent.height` **must** be `1`.
- `srcOffset.z` and `(extent.depth + srcOffset.z)` **must** both be greater than or equal to `0` and less than or equal to the source image subresource depth
- If the calling command's `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then `srcOffset.z` **must** be `0` and `extent.depth` **must** be `1`.
- `dstOffset.x` and `(extent.width + dstOffset.x)` **must** both be greater than or equal to `0` and less than or equal to the destination image subresource width
- `dstOffset.y` and `(extent.height + dstOffset.y)` **must** both be greater than or equal to `0` and less than or equal to the destination image subresource height
- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_1D`, then `dstOffset.y` **must** be `0` and `extent.height` **must** be `1`.
- `dstOffset.z` and `(extent.depth + dstOffset.z)` **must** both be greater than or equal to `0` and less than or equal to the destination image subresource depth
- If the calling command's `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then `dstOffset.z` **must** be `0` and `extent.depth` **must** be `1`.

## Valid Usage (Implicit)

- `srcSubresource` **must** be a valid `VkImageSubresourceLayers` structure
- `dstSubresource` **must** be a valid `VkImageSubresourceLayers` structure

## 19.7. Buffer Markers

To write a 32-bit marker value into a buffer as a pipelined operation, call:

```
void vkCmdWriteBufferMarkerAMD(  
    VkCommandBuffer  
    VkPipelineStageFlagBits  
    VkBuffer  
    VkDeviceSize  
    uint32_t  
                                commandBuffer,  
                                pipelineStage,  
                                dstBuffer,  
                                dstOffset,  
                                marker);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pipelineStage` is one of the `VkPipelineStageFlagBits` values, specifying the pipeline stage whose completion triggers the marker write.
- `dstBuffer` is the buffer where the marker will be written to.
- `dstOffset` is the byte offset into the buffer where the marker will be written to.
- `marker` is the 32-bit value of the marker.

The command will write the 32-bit marker value into the buffer only after all preceding commands have finished executing up to at least the specified pipeline stage. This includes the completion of other preceding `vkCmdWriteBufferMarkerAMD` commands so long as their specified pipeline stages occur either at the same time or earlier than this command's specified `pipelineStage`.

While consecutive buffer marker writes with the same `pipelineStage` parameter are implicitly complete in submission order, memory and execution dependencies between buffer marker writes and other operations must still be explicitly ordered using synchronization commands. The access scope for buffer marker writes falls under the `VK_ACCESS_TRANSFER_WRITE_BIT`, and the pipeline stages for identifying the synchronization scope **must** include both `pipelineStage` and `VK_PIPELINE_STAGE_TRANSFER_BIT`.

*Note*



Similar to `vkCmdWriteTimestamp`, if an implementation is unable to write a marker at any specific pipeline stage, it **may** instead do so at any logically later stage.

*Note*



Implementations **may** only support a limited number of pipelined marker write operations in flight at a given time, thus excessive number of marker write operations **may** degrade command execution performance.

## Valid Usage

- `dstOffset` **must** be less than or equal to the size of `dstBuffer` minus 4.
- `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag
- If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `dstOffset` **must** be a multiple of 4

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pipelineStage` **must** be a valid `VkPipelineStageFlagBits` value
- `dstBuffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- Both of `commandBuffer`, and `dstBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Transfer Graphics Compute	Transfer

# Chapter 20. Drawing Commands

*Drawing commands* (commands with `Draw` in the name) provoke work in a graphics pipeline. Drawing commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound graphics pipeline. A graphics pipeline **must** be bound to a command buffer before any drawing commands are recorded in that command buffer.

Drawing can be achieved in two modes:

- [Programmable Mesh Shading](#), the mesh shader assembles primitives, or
- [Programmable Primitive Shading](#), the input primitives are assembled

as follows.

Each draw is made up of zero or more vertices and zero or more instances, which are processed by the device and result in the assembly of primitives. Primitives are assembled according to the `pInputAssemblyState` member of the `VkGraphicsPipelineCreateInfo` structure, which is of type `VkPipelineInputAssemblyStateCreateInfo`:

```
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineInputAssemblyStateCreateFlags   flags;
    VkPrimitiveTopology        topology;
    VkBool32                  primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `topology` is a [VkPrimitiveTopology](#) defining the primitive topology, as described below.
- `primitiveRestartEnable` controls whether a special vertex index value is treated as restarting the assembly of primitives. This enable only applies to indexed draws (`vkCmdDrawIndexed` and `vkCmdDrawIndexedIndirect`), and the special index value is either `0xFFFFFFFF` when the `indexType` parameter of `vkCmdBindIndexBuffer` is equal to `VK_INDEX_TYPE_UINT32`, `0xFF` when `indexType` is equal to `VK_INDEX_TYPE_UINT8_EXT`, or `0xFFFF` when `indexType` is equal to `VK_INDEX_TYPE_UINT16`. Primitive restart is not allowed for “list” topologies.

Restarting the assembly of primitives discards the most recent index values if those elements formed an incomplete primitive, and restarts the primitive assembly using the subsequent indices, but only assembling the immediately following element through the end of the originally specified elements. The primitive restart index value comparison is performed before adding the `vertexOffset` value to the index value.

## Valid Usage

- If `topology` is `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, `primitiveRestartEnable` **must** be `VK_FALSE`
- If the `geometry shaders` feature is not enabled, `topology` **must** not be any of `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`
- If the `tessellation shaders` feature is not enabled, `topology` **must** not be `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `topology` **must** be a valid `VkPrimitiveTopology` value

```
typedef VkFlags VkPipelineInputAssemblyStateCreateFlags;
```

`VkPipelineInputAssemblyStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

## 20.1. Primitive Topologies

*Primitive topology* determines how consecutive vertices are organized into primitives, and determines the type of primitive that is used at the beginning of the graphics pipeline. The effective topology for later stages of the pipeline is altered by tessellation or geometry shading (if either is in use) and depends on the execution modes of those shaders. In the case of mesh shading the only effective topology is defined by the execution mode of the mesh shader.

The primitive topologies defined by `VkPrimitiveTopology` are:

```

typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
    VK_PRIMITIVE_TOPOLOGY_MAX_ENUM = 0x7FFFFFFF
} VkPrimitiveTopology;

```

- **VK\_PRIMITIVE\_TOPOLOGY\_POINT\_LIST** specifies a series of **separate point primitives**.
- **VK\_PRIMITIVE\_TOPOLOGY\_LINE\_LIST** specifies a series of **separate line primitives**.
- **VK\_PRIMITIVE\_TOPOLOGY\_LINE\_STRIP** specifies a series of **connected line primitives** with consecutive lines sharing a vertex.
- **VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_LIST** specifies a series of **separate triangle primitives**.
- **VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_STRIP** specifies a series of **connected triangle primitives** with consecutive triangles sharing an edge.
- **VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_FAN** specifies a series of **connected triangle primitives** with all triangles sharing a common vertex.
- **VK\_PRIMITIVE\_TOPOLOGY\_LINE\_LIST\_WITH\_ADJACENCY** specifies a series of **separate line primitives with adjacency**.
- **VK\_PRIMITIVE\_TOPOLOGY\_LINE\_STRIP\_WITH\_ADJACENCY** specifies a series of **connected line primitives with adjacency**, with consecutive primitives sharing three vertices.
- **VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_LIST\_WITH\_ADJACENCY** specifies a series of **separate triangle primitives with adjacency**.
- **VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_STRIP\_WITH\_ADJACENCY** specifies **connected triangle primitives with adjacency**, with consecutive triangles sharing an edge.
- **VK\_PRIMITIVE\_TOPOLOGY\_PATCH\_LIST** specifies **separate patch primitives**.

Each primitive topology, and its construction from a list of vertices, is described in detail below with a supporting diagram, according to the following key:

●	Vertex	A point in 3-dimensional space. Positions chosen within the diagrams are arbitrary and for illustration only.
5	Vertex Number	Sequence position of a vertex within the provided vertex data.
→	Provoking Vertex	Provoking vertex within the main primitive. The arrow points along an edge of the relevant primitive, following winding order. Used in <b>flat shading</b> .

	Primitive Edge	An edge connecting the points of a main primitive.
	Adjacency Edge	Points connected by these lines do not contribute to a main primitive, and are only accessible in a <a href="#">geometry shader</a> .
	Winding Order	The relative order in which vertices are defined within a primitive, used in the <a href="#">facing determination</a> . This ordering has no specific start or end point.

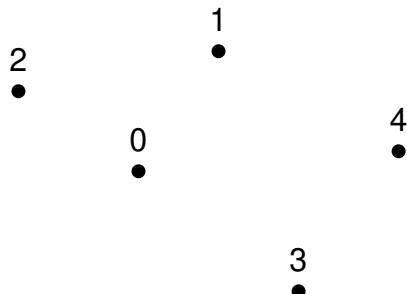
The diagrams are supported with mathematical definitions where the vertices ( $v$ ) and primitives ( $p$ ) are numbered starting from 0;  $v_0$  is the first vertex in the provided data and  $p_0$  is the first primitive in the set of primitives defined by the vertices and topology.

## 20.1.1. Point Lists

When the topology is [VK\\_PRIMITIVE\\_TOPOLOGY\\_POINT\\_LIST](#), each consecutive vertex defines a single point primitive, according to the equation:

$$p_i = \{v_i\}$$

As there is only one vertex, that vertex is the provoking vertex. The number of primitives generated is equal to [vertexCount](#).

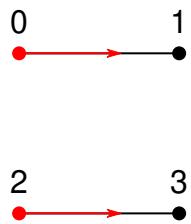


## 20.1.2. Line Lists

When the [topology](#) is [VK\\_PRIMITIVE\\_TOPOLOGY\\_LINE\\_LIST](#), each consecutive pair of vertices defines a single line primitive, according to the equation:

$$p_i = \{v_{2i}, v_{2i+1}\}$$

The provoking vertex for  $p_i$  is  $v_{2i}$ . The number of primitives generated is equal to [vertexCount/2](#).



### 20.1.3. Line Strips

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`, one line primitive is defined by each vertex and the following vertex, according to the equation:

$$p_i = \{v_i, v_{i+1}\}$$

The provoking vertex for  $p_i$  is  $v_i$ . The number of primitives generated is equal to  $\max(0, \text{vertexCount} - 1)$ .

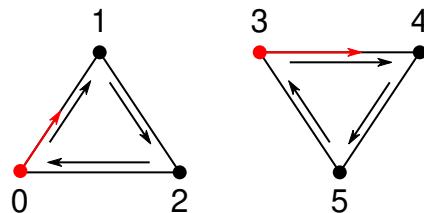


### 20.1.4. Triangle Lists

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, each consecutive set of three vertices defines a single triangle primitive, according to the equation:

$$p_i = \{v_{3i}, v_{3i+1}, v_{3i+2}\}$$

The provoking vertex for  $p_i$  is  $v_{3i}$ . The number of primitives generated is equal to  $\text{vertexCount}/3$ .

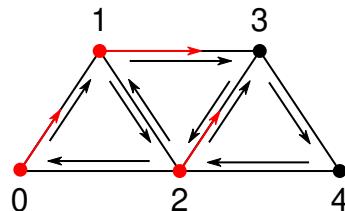


### 20.1.5. Triangle Strips

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, one triangle primitive is defined by each vertex and the two vertices that follow it, according to the equation:

$$p_i = \{v_i, v_{i+(1+i\%2)}, v_{i+(2-i\%2)}\}$$

The provoking vertex for  $p_i$  is  $v_i$ . The number of primitives generated is equal to  $\max(0, \text{vertexCount} - 2)$ .



*Note*



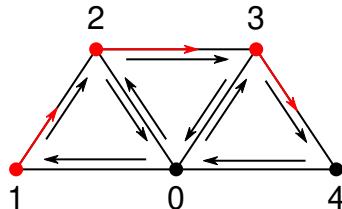
The ordering of the vertices in each successive triangle is reversed, so that the winding order is consistent throughout the strip.

## 20.1.6. Triangle Fans

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`, triangle primitives are defined around a shared common vertex, according to the equation:

$$p_i = \{v_{i+1}, v_{i+2}, v_0\}$$

The provoking vertex for  $p_i$  is  $v_{i+1}$ . The number of primitives generated is equal to  $\max(0, \text{vertexCount}-2)$ .



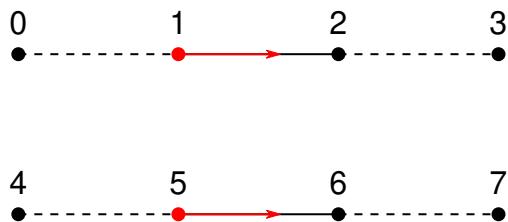
## 20.1.7. Line Lists With Adjacency

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, each consecutive set of four vertices defines a single line primitive with adjacency, according to the equation:

$$p_i = \{v_{4i}, v_{4i+1}, v_{4i+2}, v_{4i+3}\}$$

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a `geometry shader`.

The provoking vertex for  $p_i$  is  $v_{4i+1}$ . The number of primitives generated is equal to `vertexCount/4`.



## 20.1.8. Line Strips With Adjacency

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`, one line primitive with adjacency is defined by each vertex and the following vertex, according to the equation:

$$p_i = \{v_i, v_{i+1}, v_{i+2}, v_{i+3}\}$$

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a `geometry shader`.

The provoking vertex for  $p_i$  is  $v_{i+1}$ . The number of primitives generated is equal to  $\max(0, \text{vertexCount}-3)$ .



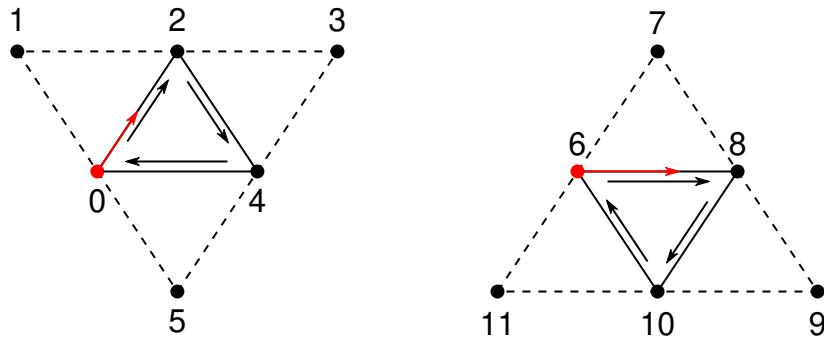
## 20.1.9. Triangle Lists With Adjacency

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY`, each consecutive set of six vertices defines a single triangle primitive with adjacency, according to the equations:

$$p_i = \{v_{6i}, v_{6i+1}, v_{6i+2}, v_{6i+3}, v_{6i+4}, v_{6i+5}\}$$

A triangle primitive is described by the first, third, and fifth vertices of the total primitive, with the remaining three vertices only accessible in a [geometry shader](#).

The provoking vertex for  $p_i$  is  $v_{6i}$ . The number of primitives generated is equal to `vertexCount/6` .



## 20.1.10. Triangle Strips With Adjacency

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`, one triangle primitive with adjacency is defined by each vertex and the following 5 vertices.

The number of primitives generated,  $n$ , is equal to `max(0, vertexCount - 4)/2` .

If  $n=1$ , the primitive is defined as:

$$p_i = \{v_0, v_1, v_2, v_5, v_4, v_3\}$$

If  $n>1$ , the total primitive consists of different vertices according to where it is in the strip:

$$p_i = \{v_{2i}, v_{2i+1}, v_{2i+2}, v_{2i+6}, v_{2i+4}, v_{2i+3}\} \text{ when } i=0$$

$$p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+6}, v_{2i+2}, v_{2i+1}\} \text{ when } i>0, i<n-1, \text{ and } i \% 2 = 1$$

$$p_i = \{v_{2i}, v_{2i-2}, v_{2i+2}, v_{2i+6}, v_{2i+4}, v_{2i+3}\} \text{ when } i>0, i<n-1, \text{ and } i \% 2 = 0$$

$$p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+5}, v_{2i+2}, v_{2i-2}\} \text{ when } i=n-1 \text{ and } i \% 2 = 1$$

$$p_i = \{v_{2i}, v_{2i-2}, v_{2i+2}, v_{2i+5}, v_{2i+4}, v_{2i+3}\} \text{ when } i=n-1 \text{ and } i \% 2 = 0$$

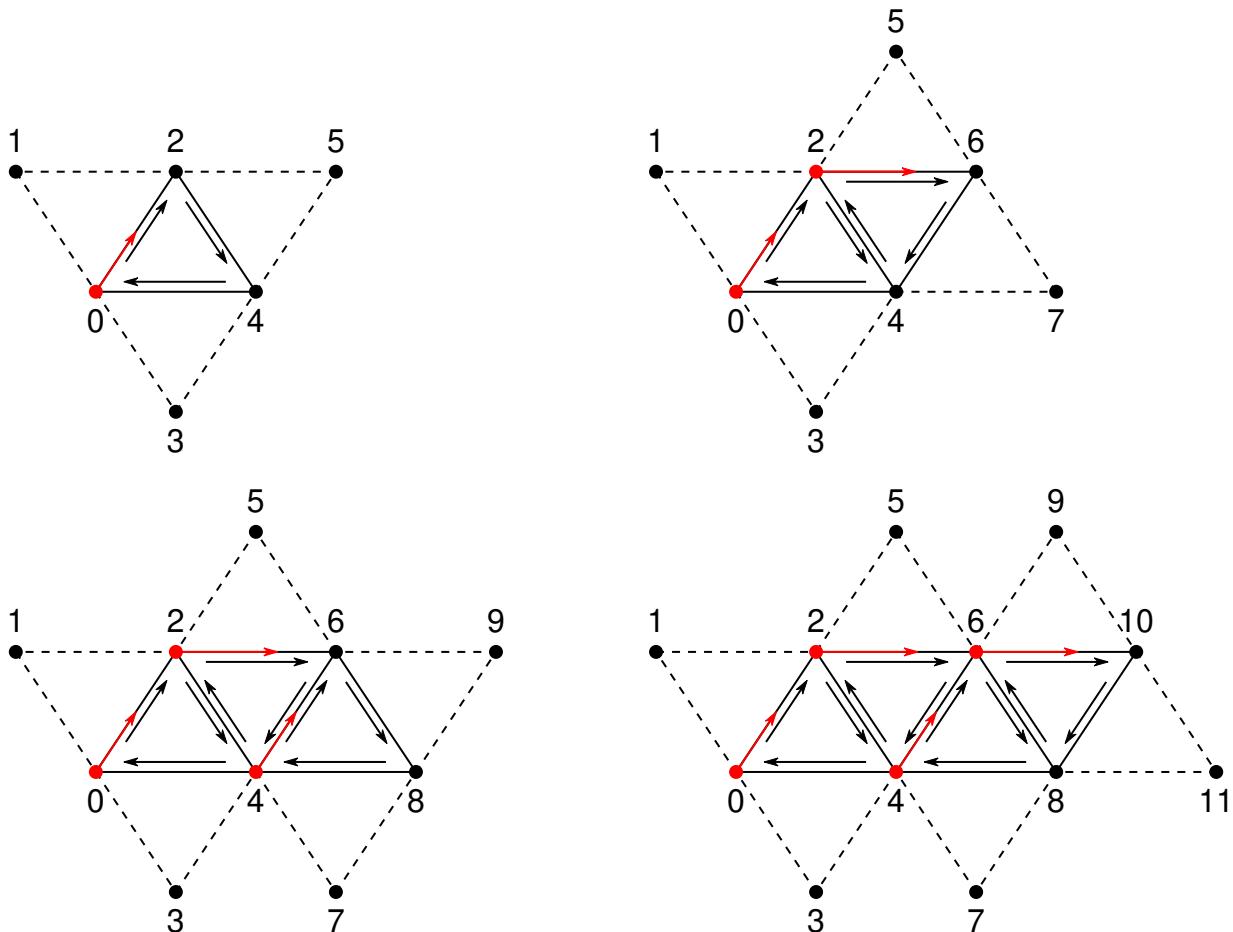
A triangle primitive is described by the first, third, and fifth vertices of the total primitive in all cases, with the remaining three vertices only accessible in a [geometry shader](#).

### Note



The ordering of the vertices in each successive triangle is altered so that the winding order is consistent throughout the strip.

The provoking vertex for  $p_i$  is always  $v_{2i}$ .



### 20.1.11. Patch Lists

When the [topology](#) is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, each consecutive set of  $m$  vertices defines a single patch primitive, according to the equation:

$$p_i = \{v_{mi}, v_{mi+1}, \dots, v_{mi+(m-2)}, v_{mi+(m-1)}\}$$

where  $m$  is equal to [`VkPipelineTessellationStateCreateInfo::patchControlPoints`](#).

Patch lists are never passed to [vertex post-processing](#), and as such no provoking vertex is defined for patch primitives. The number of primitives generated is equal to  $\text{vertexCount}/m$  .

The vertices comprising a patch have no implied geometry, and are used as inputs to tessellation shaders and the fixed-function tessellator to generate new point, line, or triangle primitives.

## 20.2. Primitive Order

Primitives generated by [drawing commands](#) progress through the stages of the [graphics pipeline](#) in *primitive order*. Primitive order is initially determined in the following way:

1. Submission order determines the initial ordering
2. For indirect draw commands, the order in which accessed instances of the `VkDrawIndirectCommand` are stored in `buffer`, from lower indirect buffer addresses to higher addresses.
3. If a draw command includes multiple instances, the order in which instances are executed, from lower numbered instances to higher.
4. The order in which primitives are specified by a draw command:
  - For non-indexed draws, from vertices with a lower numbered `vertexIndex` to a higher numbered `vertexIndex`.
  - For indexed draws, vertices sourced from a lower index buffer addresses to higher addresses.
  - For draws using mesh shaders, the order is provided by `mesh shading`.

Within this order implementations further sort primitives:

5. If tessellation shading is active, by an implementation-dependent order of new primitives generated by `tessellation`.
6. If geometry shading is active, by the order new primitives are generated by `geometry shading`.
7. If the `polygon mode` is not `VK_POLYGON_MODE_FILL`, or `VK_POLYGON_MODE_FILL_RECTANGLE_NV`, by an implementation-dependent ordering of the new primitives generated within the original primitive.

Primitive order is later used to define `rasterization order`, which determines the order in which fragments output results to a framebuffer.

## 20.3. Programmable Primitive Shading

Once primitives are assembled, they proceed to the vertex shading stage of the pipeline. If the draw includes multiple instances, then the set of primitives is sent to the vertex shading stage multiple times, once for each instance.

It is implementation-dependent whether vertex shading occurs on vertices that are discarded as part of incomplete primitives, but if it does occur then it operates as if they were vertices in complete primitives and such invocations **can** have side effects.

Vertex shading receives two per-vertex inputs from the primitive assembly stage - the `vertexIndex` and the `instanceIndex`. How these values are generated is defined below, with each command.

Drawing commands fall roughly into two categories:

- Non-indexed drawing commands present a sequential `vertexIndex` to the vertex shader. The sequential index is generated automatically by the device (see [Fixed-Function Vertex Processing](#) for details on both specifying the vertex attributes indexed by `vertexIndex`, as well as binding vertex buffers containing those attributes to a command buffer). These commands are:
  - `vkCmdDraw`

- `vkCmdDrawIndirect`
- `vkCmdDrawIndirectCountKHR`
- `vkCmdDrawIndirectCountAMD`
- Indexed drawing commands read index values from an *index buffer* and use this to compute the `vertexIndex` value for the vertex shader. These commands are:
  - `vkCmdDrawIndexed`
  - `vkCmdDrawIndexedIndirect`
  - `vkCmdDrawIndexedIndirectCountKHR`
  - `vkCmdDrawIndexedIndirectCountAMD`

To bind an index buffer to a command buffer, call:

```
void vkCmdBindIndexBuffer(
    VkCommandBuffer           commandBuffer,
    VkBuffer                  buffer,
    VkDeviceSize              offset,
    VkIndexType               indexType);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer being bound.
- `offset` is the starting offset in bytes within `buffer` used in index buffer address calculations.
- `indexType` is a `VkIndexType` value specifying whether indices are treated as 16 bits or 32 bits.

## Valid Usage

- `offset` **must** be less than the size of `buffer`
- The sum of `offset` and the address of the range of `VkDeviceMemory` object that is backing `buffer`, **must** be a multiple of the type indicated by `indexType`
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` flag
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `indexType` **must** not be `VK_INDEX_TYPE_NONE_NV`.
- If `indexType` is `VK_INDEX_TYPE_UINT8_EXT`, the `indexTypeUint8` feature **must** be enabled

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `indexType` **must** be a valid `VkIndexType` value
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- Both of `buffer`, and `commandBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

Possible values of `vkCmdBindIndexBuffer::indexType`, specifying the size of indices, are:

```
typedef enum VkIndexType {
    VK_INDEX_TYPE_UINT16 = 0,
    VK_INDEX_TYPE_UINT32 = 1,
    VK_INDEX_TYPE_NONE_NV = 1000165000,
    VK_INDEX_TYPE_UINT8_EXT = 1000265000,
    VK_INDEX_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkIndexType;
```

- `VK_INDEX_TYPE_UINT16` specifies that indices are 16-bit unsigned integer values.
- `VK_INDEX_TYPE_UINT32` specifies that indices are 32-bit unsigned integer values.
- `VK_INDEX_TYPE_NONE_NV` specifies that no indices are provided.
- `VK_INDEX_TYPE_UINT8_EXT` specifies that indices are 8-bit unsigned integer values.

The parameters for each drawing command are specified directly in the command or read from

buffer memory, depending on the command. Drawing commands that source their parameters from buffer memory are known as *indirect* drawing commands.

All drawing commands interact with the [Robust Buffer Access](#) feature.

To record a non-indexed draw, call:

```
void vkCmdDraw(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
        commandBuffer,  
        vertexCount,  
        instanceCount,  
        firstVertex,  
        firstInstance);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `vertexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstVertex` is the index of the first vertex to draw.
- `firstInstance` is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and `vertexCount` consecutive vertex indices with the first `vertexIndex` value equal to `firstVertex`. The primitives are drawn `instanceCount` times with `instanceIndex` starting with `firstInstance` and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- If `commandBuffer` is a protected command buffer, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be an unprotected resource
- If `commandBuffer` is a protected command buffer, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point **must** not write to any resource
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound

- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

To record an indexed draw, call:

```
void vkCmdDrawIndexed(
    VkCommandBuffer
    uint32_t
    uint32_t
    uint32_t
    int32_t
    uint32_t
        commandBuffer,
        indexCount,
        instanceCount,
        firstIndex,
        vertexOffset,
        firstInstance);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `indexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstIndex` is the base index within the index buffer.
- `vertexOffset` is the value added to the vertex index before indexing into the vertex buffer.
- `firstInstance` is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and `indexCount` vertices whose indices are retrieved from the index buffer. The index buffer is treated as an array of tightly packed unsigned integers of size defined by the `vkCmdBindIndexBuffer` ::`indexType` parameter with which the buffer was bound.

The first vertex index is at an offset of `firstIndex * indexSize + offset` within the bound index buffer, where `offset` is the offset specified by `vkCmdBindIndexBuffer` and `indexSize` is the byte size of the type specified by `indexType`. Subsequent index values are retrieved from consecutive locations in the index buffer. Indices are first compared to the primitive restart value, then zero extended to 32 bits (if the `indexType` is `VK_INDEX_TYPE_UINT8_EXT` or `VK_INDEX_TYPE_UINT16`) and have `vertexOffset` added to them, before being supplied as the `vertexIndex` value.

The primitives are drawn `instanceCount` times with `instanceIndex` starting with `firstInstance` and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- If `commandBuffer` is a protected command buffer, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be an unprotected resource
- If `commandBuffer` is a protected command buffer, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point **must** not write to any resource
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound

- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- $(\text{indexSize} * (\text{firstIndex} + \text{indexCount}) + \text{offset})$  **must** be less than or equal to the size of the bound index buffer, with `indexSize` being based on the type specified by `indexType`, where the index buffer, `indexType`, and `offset` are specified via `vkCmdBindIndexBuffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the [recording state](#)
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Inside	Graphics	Graphics
Secondary			

To record a non-indexed indirect draw, call:

```
void vkCmdDrawIndirect(
    VkCommandBuffer
    VkBuffer
    VkDeviceSize
    uint32_t
    uint32_t
        commandBuffer,
        buffer,
        offset,
        drawCount,
        stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `drawCount` is the number of draws to execute, and **can** be zero.

- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndirect` behaves similarly to `vkCmdDraw` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawIndirectCommand` structures. If `drawCount` is less than or equal to one, `stride` is ignored.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set

- `offset` **must** be a multiple of 4
- `commandBuffer` **must** not be a protected command buffer
- If the `multi-draw indirect` feature is not enabled, `drawCount` **must** be 0 or 1
- `drawCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`
- If the `drawIndirectFirstInstance` feature is not enabled, all the `firstInstance` members of the `VkDrawIndirectCommand` structures accessed by this command **must** be 0
- If `drawCount` is greater than 1, `stride` **must** be a multiple of 4 and **must** be greater than or equal to `sizeof(VkDrawIndirectCommand)`
- If `drawCount` is equal to 1, `(offset + sizeof(VkDrawIndirectCommand))` **must** be less than or equal to the size of `buffer`
- If `drawCount` is greater than 1, `(stride × (drawCount - 1) + offset + sizeof(VkDrawIndirectCommand))` **must** be less than or equal to the size of `buffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Both of `buffer`, and `commandBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

The `VkDrawIndirectCommand` structure is defined as:

```
typedef struct VkDrawIndirectCommand {  
    uint32_t    vertexCount;  
    uint32_t    instanceCount;  
    uint32_t    firstVertex;  
    uint32_t    firstInstance;  
} VkDrawIndirectCommand;
```

- `vertexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstVertex` is the index of the first vertex to draw.
- `firstInstance` is the instance ID of the first instance to draw.

The members of `VkDrawIndirectCommand` have the same meaning as the similarly named parameters of `vkCmdDraw`.

## Valid Usage

- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- If the `drawIndirectFirstInstance` feature is not enabled, `firstInstance` **must** be `0`

To record a non-indexed draw call with a draw call count sourced from a buffer, call:

```
void vkCmdDrawIndirectCountKHR(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
    VkBuffer  
    VkDeviceSize  
    uint32_t  
    uint32_t  
        commandBuffer,  
        buffer,  
        offset,  
        countBuffer,  
        countBufferOffset,  
        maxDrawCount,  
        stride);
```

or the equivalent command

```
void vkCmdDrawIndirectCountAMD(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
    VkBuffer  
    VkDeviceSize  
    uint32_t  
    uint32_t  
        commandBuffer,  
        buffer,  
        offset,  
        countBuffer,  
        countBufferOffset,  
        maxDrawCount,  
        stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.

- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `countBuffer` is the buffer containing the draw count.
- `countBufferOffset` is the byte offset into `countBuffer` where the draw count begins.
- `maxDrawCount` specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in `countBuffer` and `maxDrawCount`.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndirectCountKHR` behaves similarly to `vkCmdDrawIndirect` except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from `countBuffer` located at `countBufferOffset` and use this as the draw count.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set

- `offset` **must** be a multiple of 4
- `commandBuffer` **must** not be a protected command buffer
- If `countBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `countBuffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `countBufferOffset` **must** be a multiple of 4
- The count stored in `countBuffer` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`
- `stride` **must** be a multiple of 4 and **must** be greater than or equal to `sizeof(VkDrawIndirectCommand)`
- If `maxDrawCount` is greater than or equal to 1,  $(\text{stride} \times (\text{maxDrawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawIndirectCommand}))$  **must** be less than or equal to the size of `buffer`
- If the count stored in `countBuffer` is equal to 1,  $(\text{offset} + \text{sizeof}(\text{VkDrawIndirectCommand}))$  **must** be less than or equal to the size of `buffer`
- If the count stored in `countBuffer` is greater than 1,  $(\text{stride} \times (\text{drawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawIndirectCommand}))$  **must** be less than or equal to the size of `buffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `countBuffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Each of `buffer`, `commandBuffer`, and `countBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

To record an indexed indirect draw, call:

```
void vkCmdDrawIndexedIndirect(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
    uint32_t  
    uint32_t  
                                commandBuffer,  
                                buffer,  
                                offset,  
                                drawCount,  
                                stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `drawCount` is the number of draws to execute, and `can` be zero.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndexedIndirect` behaves similarly to `vkCmdDrawIndexed` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawIndexedIndirectCommand` structures. If `drawCount` is less than or equal to one, `stride` is ignored.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set

- `offset` **must** be a multiple of 4
- `commandBuffer` **must** not be a protected command buffer
- If the `multi-draw indirect` feature is not enabled, `drawCount` **must** be 0 or 1
- `drawCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`
- If `drawCount` is greater than 1, `stride` **must** be a multiple of 4 and **must** be greater than or equal to `sizeof(VkDrawIndexedIndirectCommand)`
- If the `drawIndirectFirstInstance` feature is not enabled, all the `firstInstance` members of the `VkDrawIndexedIndirectCommand` structures accessed by this command **must** be 0
- If `drawCount` is equal to 1, `(offset + sizeof(VkDrawIndexedIndirectCommand))` **must** be less than or equal to the size of `buffer`
- If `drawCount` is greater than 1, `(stride × (drawCount - 1) + offset + sizeof(VkDrawIndexedIndirectCommand))` **must** be less than or equal to the size of `buffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Both of `buffer`, and `commandBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

The `VkDrawIndexedIndirectCommand` structure is defined as:

```

typedef struct VkDrawIndexedIndirectCommand {
    uint32_t    indexCount;
    uint32_t    instanceCount;
    uint32_t    firstIndex;
    int32_t     vertexOffset;
    uint32_t    firstInstance;
} VkDrawIndexedIndirectCommand;

```

- `indexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstIndex` is the base index within the index buffer.
- `vertexOffset` is the value added to the vertex index before indexing into the vertex buffer.
- `firstInstance` is the instance ID of the first instance to draw.

The members of `VkDrawIndexedIndirectCommand` have the same meaning as the similarly named parameters of `vkCmdDrawIndexed`.

## Valid Usage

- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- `(indexSize * (firstIndex + indexCount) + offset)` **must** be less than or equal to the size of the bound index buffer, with `indexSize` being based on the type specified by `indexType`, where the index buffer, `indexType`, and `offset` are specified via `vkCmdBindIndexBuffer`
- If the `drawIndirectFirstInstance` feature is not enabled, `firstInstance` **must** be `0`

To record an indexed draw call with a draw call count sourced from a buffer, call:

```

void vkCmdDrawIndexedIndirectCountKHR(
    VkCommandBuffer                                commandBuffer,
    VkBuffer                                         buffer,
    VkDeviceSize                                     offset,
    VkBuffer                                         countBuffer,
    VkDeviceSize                                     countBufferOffset,
    uint32_t                                         maxDrawCount,
    uint32_t                                         stride);

```

or the equivalent command

```
void vkCmdDrawIndexedIndirectCountAMD(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
    VkBuffer  
    VkDeviceSize  
    uint32_t  
    uint32_t  
                                commandBuffer,  
                                buffer,  
                                offset,  
                                countBuffer,  
                                countBufferOffset,  
                                maxDrawCount,  
                                stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `countBuffer` is the buffer containing the draw count.
- `countBufferOffset` is the byte offset into `countBuffer` where the draw count begins.
- `maxDrawCount` specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in `countBuffer` and `maxDrawCount`.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndexedIndirectCountKHR` behaves similarly to `vkCmdDrawIndexedIndirect` except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from `countBuffer` located at `countBufferOffset` and use this as the draw count.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set

- `offset` **must** be a multiple of 4
- `commandBuffer` **must** not be a protected command buffer
- If `countBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `countBuffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `countBufferOffset` **must** be a multiple of 4
- The count stored in `countBuffer` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`
- `stride` **must** be a multiple of 4 and **must** be greater than or equal to `sizeof(VkDrawIndexedIndirectCommand)`
- If `maxDrawCount` is greater than or equal to 1,  $(\text{stride} \times (\text{maxDrawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawIndexedIndirectCommand}))$  **must** be less than or equal to the size of `buffer`
- If count stored in `countBuffer` is equal to 1,  $(\text{offset} + \text{sizeof}(\text{VkDrawIndexedIndirectCommand}))$  **must** be less than or equal to the size of `buffer`
- If count stored in `countBuffer` is greater than 1,  $(\text{stride} \times (\text{drawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawIndexedIndirectCommand}))$  **must** be less than or equal to the size of `buffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `countBuffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Each of `buffer`, `commandBuffer`, and `countBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

### 20.3.1. Drawing Transform Feedback

It is possible to draw vertex data that was previously captured during active [transform feedback](#) by binding one or more of the transform feedback buffers as vertex buffers. A pipeline barrier is required between using the buffers as transform feedback buffers and vertex buffers to ensure all writes to the transform feedback buffers are visible when the data is read as vertex attributes. The source access is `VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT` and the destination access is `VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT` for the pipeline stages `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT` and `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` respectively. The value written to the counter buffer by `vkCmdEndTransformFeedbackEXT` can be used to determine the vertex count for the draw. A pipeline barrier is required between using the counter buffer for `vkCmdEndTransformFeedbackEXT` and `vkCmdDrawIndirectByteCountEXT` where the source access is `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT` and the destination access is `VK_ACCESS_INDIRECT_COMMAND_READ_BIT` for the pipeline stages `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT` and `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT` respectively.

To record a non-indexed draw call, where the vertex count is based on a byte count read from a buffer and the passed in vertex stride parameter, call:

```
void vkCmdDrawIndirectByteCountEXT(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    VkBuffer  
    VkDeviceSize  
    uint32_t  
    uint32_t  
    commandBuffer,  
    instanceCount,  
    firstInstance,  
    counterBuffer,  
    counterBufferOffset,  
    counterOffset,  
    vertexStride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `instanceCount` is the number of instances to draw.
- `firstInstance` is the instance ID of the first instance to draw.
- `counterBuffer` is the buffer handle from where the byte count is read.
- `counterBufferOffset` is the offset into the buffer used to read the byte count, which is used to calculate the vertex count for this draw call.
- `counterOffset` is subtracted from the byte count read from the `counterBuffer` at the `counterBufferOffset`

- `vertexStride` is the stride in bytes between each element of the vertex data that is used to calculate the vertex count from the counter value. This value is typically the same value that was used in the graphics pipeline state when the transform feedback was captured as the `XfbStride`.

When the command is executed, primitives are assembled in the same way as done with `vkCmdDraw` except the `vertexCount` is calculated based on the byte count read from `counterBuffer` at offset `counterBufferOffset`. The assembled primitives execute the bound graphics pipeline.

The effective `vertexCount` is calculated as follows:

```
const uint32_t * counterBufferPtr = (const uint8_t *)counterBuffer.address +  
counterBufferOffset;  
vertexCount = floor(max(0, (*counterBufferPtr - counterOffset)) / vertexStride);
```

The effective `firstVertex` is zero.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in [Vertex Input Description](#)
- `VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback` **must** be enabled
- The implementation **must** support `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackDraw`

- `vertexStride` **must** be greater than 0 and less than or equal to `VkPhysicalDeviceLimits::maxTransformFeedbackBufferDataStride`
- `counterBuffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `commandBuffer` **must** not be a protected command buffer

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `counterBuffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Both of `commandBuffer`, and `counterBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

## 20.4. Conditional Rendering

Certain rendering commands **can** be executed conditionally based on a value in buffer memory. These rendering commands are limited to `drawing commands`, `dispatching commands`, and clearing attachments with `vkCmdClearAttachments` within a conditional rendering block which is defined by commands `vkCmdBeginConditionalRenderingEXT` and `vkCmdEndConditionalRenderingEXT`. Other rendering commands remain unaffected by conditional rendering.

After beginning conditional rendering, it is considered *active* within the command buffer it was

called until it is ended with [vkCmdEndConditionalRenderingEXT](#).

Conditional rendering **must** begin and end in the same command buffer. When conditional rendering is active, a primary command buffer **can** execute secondary command buffers if the [inherited conditional rendering](#) feature is enabled. For a secondary command buffer to be executed while conditional rendering is active in the primary command buffer, it **must** set the [conditionalRenderingEnable](#) flag of [VkCommandBufferInheritanceConditionalRenderingInfoEXT](#), as described in the [Command Buffer Recording](#) section.

Conditional rendering **must** also either begin and end inside the same subpass of a render pass instance, or **must** both begin and end outside of a render pass instance (i.e. contain entire render pass instances).

To begin conditional rendering, call:

```
void vkCmdBeginConditionalRenderingEXT(  
    VkCommandBuffer  
        commandBuffer,  
    const VkConditionalRenderingBeginInfoEXT*  
        pConditionalRenderingBegin);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `pConditionalRenderingBegin` is a pointer to a [VkConditionalRenderingBeginInfoEXT](#) structure specifying parameters of conditional rendering.

### Valid Usage

- Conditional rendering **must** not already be [active](#)

### Valid Usage (Implicit)

- `commandBuffer` **must** be a valid [VkCommandBuffer](#) handle
- `pConditionalRenderingBegin` **must** be a valid pointer to a valid [VkConditionalRenderingBeginInfoEXT](#) structure
- `commandBuffer` **must** be in the [recording state](#)
- The [VkCommandPool](#) that `commandBuffer` was allocated from **must** support graphics, or compute operations

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the [VkCommandPool](#) that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

The `VkConditionalRenderingBeginInfoEXT` structure is defined as:

```
typedef struct VkConditionalRenderingBeginInfoEXT {
    VkStructureType          sType;
    const void*               pNext;
    VkBuffer                  buffer;
    VkDeviceSize               offset;
    VkConditionalRenderingFlagsEXT flags;
} VkConditionalRenderingBeginInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `buffer` is a buffer containing the predicate for conditional rendering.
- `offset` is the byte offset into `buffer` where the predicate is located.
- `flags` is a bitmask of `VkConditionalRenderingFlagsEXT` specifying the behavior of conditional rendering.

If the 32-bit value at `offset` in `buffer` memory is zero, then the rendering commands are discarded, otherwise they are executed as normal. If the value of the predicate in buffer memory changes while conditional rendering is active, the rendering commands **may** be discarded in an implementation-dependent way. Some implementations may latch the value of the predicate upon beginning conditional rendering while others may read it before every rendering command.

## Valid Usage

- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_CONDITIONAL_RENDERING_BIT_EXT` bit set
- `offset` **must** be less than the size of `buffer` by at least 32 bits.
- `offset` **must** be a multiple of 4

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_CONDITIONAL_RENDERING_BEGIN_INFO_EXT`
- `pNext` **must** be `NULL`
- `buffer` **must** be a valid `VkBuffer` handle
- `flags` **must** be a valid combination of `VkConditionalRenderingFlagBitsEXT` values

Bits which **can** be set in `vkCmdBeginConditionalRenderingEXT::flags` specifying the behavior of conditional rendering are:

```
typedef enum VkConditionalRenderingFlagBitsEXT {
    VK_CONDITIONAL_RENDERING_INVERTED_BIT_EXT = 0x00000001,
    VK_CONDITIONAL_RENDERING_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkConditionalRenderingFlagBitsEXT;
```

- `VK_CONDITIONAL_RENDERING_INVERTED_BIT_EXT` specifies the condition used to determine whether to discard rendering commands or not. That is, if the 32-bit predicate read from `buffer` memory at `offset` is zero, the rendering commands are not discarded, and if non zero, then they are discarded.

```
typedef VkFlags VkConditionalRenderingFlagsEXT;
```

`VkConditionalRenderingFlagsEXT` is a bitmask type for setting a mask of zero or more `VkConditionalRenderingFlagBitsEXT`.

To end conditional rendering, call:

```
void vkCmdEndConditionalRendering(
    VkCommandBuffer               commandBuffer);
```

- `commandBuffer` is the command buffer into which this command will be recorded.

Once ended, conditional rendering becomes inactive.

## Valid Usage

- Conditional rendering **must** be `active`
- If conditional rendering was made `active` outside of a render pass instance, it must not be ended inside a render pass instance
- If conditional rendering was made `active` within a subpass it must be ended in the same subpass

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

## 20.5. Programmable Mesh Shading

In this drawing approach, primitives are assembled by the mesh shader stage. `Mesh shading` operates similarly to `dispatching compute` as the shaders make use of workgroups.

To record a draw that uses the mesh pipeline, call:

```
void vkCmdDrawMeshTasksNV(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
                                commandBuffer,  
                                taskCount,  
                                firstTask);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `taskCount` is the number of local workgroups to dispatch in the X dimension. Y and Z dimension are implicitly set to one.
- `firstTask` is the X component of the first workgroup ID.

When the command is executed, a global workgroup consisting of `taskCount` local workgroups is assembled.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- `taskCount` **must** be less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxDrawMeshTasksCount`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

To record an indirect mesh tasks draw, call:

```
void vkCmdDrawMeshTasksIndirectNV(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
    uint32_t  
    uint32_t  
        commandBuffer,  
        buffer,  
        offset,  
        drawCount,  
        stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `drawCount` is the number of draws to execute, and **can** be zero.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawMeshTasksIndirectNV` behaves similarly to `vkCmdDrawMeshTasksNV` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawMeshTasksIndirectCommandNV` structures. If `drawCount` is less than or equal to one, `stride`

is ignored.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `offset` **must** be a multiple of 4
- `commandBuffer` **must** not be a protected command buffer
- If the `multi-draw indirect` feature is not enabled, `drawCount` **must** be 0 or 1
- `drawCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`

- If `drawCount` is greater than 1, `stride` must be a multiple of 4 and must be greater than or equal to `sizeof(VkDrawMeshTasksIndirectCommandNV)`
- If `drawCount` is equal to 1, `(offset + sizeof(VkDrawMeshTasksIndirectCommandNV))` must be less than or equal to the size of `buffer`
- If `drawCount` is greater than 1, `(stride × (drawCount - 1) + offset + sizeof(VkDrawMeshTasksIndirectCommandNV))` must be less than or equal to the size of `buffer`

## Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `buffer` must be a valid `VkBuffer` handle
- `commandBuffer` must be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- Both of `buffer`, and `commandBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

The `VkDrawMeshTasksIndirectCommandNV` structure is defined as:

```
typedef struct VkDrawMeshTasksIndirectCommandNV {
    uint32_t taskCount;
    uint32_t firstTask;
} VkDrawMeshTasksIndirectCommandNV;
```

- `taskCount` is the number of local workgroups to dispatch in the X dimension. Y and Z dimension are implicitly set to one.

- `firstTask` is the X component of the first workgroup ID.

The members of `VkDrawMeshTasksIndirectCommandNV` have the same meaning as the similarly named parameters of `vkCmdDrawMeshTasksNV`.

## Valid Usage

- `taskCount` **must** be less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxDrawMeshTasksCount`

To record an indirect mesh tasks draw with the draw count sourced from a buffer, call:

```
void vkCmdDrawMeshTasksIndirectCountNV(
    VkCommandBuffer                                commandBuffer,
    VkBuffer                                         buffer,
    VkDeviceSize                                     offset,
    VkBuffer                                         countBuffer,
    VkDeviceSize                                     countBufferOffset,
    uint32_t                                         maxDrawCount,
    uint32_t                                         stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `countBuffer` is the buffer containing the draw count.
- `countBufferOffset` is the byte offset into `countBuffer` where the draw count begins.
- `maxDrawCount` specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in `countBuffer` and `maxDrawCount`.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawMeshTasksIndirectCountNV` behaves similarly to `vkCmdDrawMeshTasksIndirectNV` except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from `countBuffer` located at `countBufferOffset` and use this as the draw count.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- The current render pass **must** be `compatible` with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command.
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.
- If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `offset` **must** be a multiple of 4
- `commandBuffer` **must** not be a protected command buffer
- If `countBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- `countBuffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `countBufferOffset` **must** be a multiple of 4
- The count stored in `countBuffer` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`
- `stride` **must** be a multiple of 4 and **must** be greater than or equal to `sizeof(VkDrawMeshTasksIndirectCommandNV)`
- If `maxDrawCount` is greater than or equal to 1,  $(\text{stride} \times (\text{maxDrawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawMeshTasksIndirectCommandNV}))$  **must** be less than or equal to the size of `buffer`
- If the count stored in `countBuffer` is equal to 1,  $(\text{offset} + \text{sizeof}(\text{VkDrawMeshTasksIndirectCommandNV}))$  **must** be less than or equal to the size of `buffer`
- If the count stored in `countBuffer` is greater than 1,  $(\text{stride} \times (\text{drawCount} - 1) + \text{offset} + \text{sizeof}(\text{VkDrawMeshTasksIndirectCommandNV}))$  **must** be less than or equal to the size of `buffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `countBuffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Each of `buffer`, `commandBuffer`, and `countBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	Graphics

# Chapter 21. Fixed-Function Vertex Processing

Vertex fetching is controlled via configurable state, as a logically distinct graphics pipeline stage.

## 21.1. Vertex Attributes

Vertex shaders **can** define input variables, which receive *vertex attribute* data transferred from one or more `VkBuffer`(s) by drawing commands. Vertex shader input variables are bound to buffers via an indirect binding where the vertex shader associates a *vertex input attribute* number with each variable, vertex input attributes are associated to *vertex input bindings* on a per-pipeline basis, and vertex input bindings are associated with specific buffers on a per-draw basis via the `vkCmdBindVertexBuffers` command. Vertex input attribute and vertex input binding descriptions also contain format information controlling how data is extracted from buffer memory and converted to the format expected by the vertex shader.

There are `VkPhysicalDeviceLimits::maxVertexInputAttributes` number of vertex input attributes and `VkPhysicalDeviceLimits::maxVertexInputBindings` number of vertex input bindings (each referred to by zero-based indices), where there are at least as many vertex input attributes as there are vertex input bindings. Applications **can** store multiple vertex input attributes interleaved in a single buffer, and use a single vertex input binding to access those attributes.

In GLSL, vertex shaders associate input variables with a vertex input attribute number using the `location` layout qualifier. The `component` layout qualifier associates components of a vertex shader input variable with components of a vertex input attribute.

*GLSL example*

```
// Assign location M to variableName
layout (location=M, component=2) in vec2 variableName;

// Assign locations [N,N+L) to the array elements of variableNameArray
layout (location=N) in vec4 variableNameArray[L];
```

In SPIR-V, vertex shaders associate input variables with a vertex input attribute number using the `Location` decoration. The `Component` decoration associates components of a vertex shader input variable with components of a vertex input attribute. The `Location` and `Component` decorations are specified via the `OpDecorate` instruction.

```

...
%1 = OpExtInstImport "GLSL.std.450"
...
OpName %9 "variableName"
OpName %15 "variableNameArray"
OpDecorate %18 BuiltIn VertexIndex
OpDecorate %19 BuiltIn InstanceIndex
OpDecorate %9 Location M
OpDecorate %9 Component 2
OpDecorate %15 Location N
...
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeVector %6 2
%8 = OpTypePointer Input %7
%9 = OpVariable %8 Input
%10 = OpTypeVector %6 4
%11 = OpTypeInt 32 0
%12 = OpConstant %11 L
%13 = OpTypeArray %10 %12
%14 = OpTypePointer Input %13
%15 = OpVariable %14 Input
...

```

### 21.1.1. Attribute Location and Component Assignment

Vertex shaders allow **Location** and **Component** decorations on input variable declarations. The **Location** decoration specifies which vertex input attribute is used to read and interpret the data that a variable will consume. The **Component** decoration allows the location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable starting at component N will consume components N, N+1, N+2, ... up through its size. For single precision types, it is invalid if the sequence of components gets larger than 3.

When a vertex shader input variable declared using a scalar or vector 32-bit data type is assigned a location, its value(s) are taken from the components of the input attribute specified with the corresponding `VkVertexInputAttributeDescription::location`. The components used depend on the type of variable and the **Component** decoration specified in the variable declaration, as identified in [Input attribute components accessed by 32-bit input variables](#). Any 32-bit scalar or vector input will consume a single location. For 32-bit data types, missing components are filled in with default values as described [below](#).

*Table 29. Input attribute components accessed by 32-bit input variables*

<b>32-bit data type</b>	<b>Component decoration</b>	<b>Components consumed</b>
scalar	0 or unspecified	(x, o, o, o)
scalar	1	(o, y, o, o)
scalar	2	(o, o, z, o)
scalar	3	(o, o, o, w)
two-component vector	0 or unspecified	(x, y, o, o)
two-component vector	1	(o, y, z, o)
two-component vector	2	(o, o, z, w)
three-component vector	0 or unspecified	(x, y, z, o)
three-component vector	1	(o, y, z, w)
four-component vector	0 or unspecified	(x, y, z, w)

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input format (if present), or the default value.

When a vertex shader input variable declared using a 32-bit floating point matrix type is assigned a location  $i$ , its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. Such matrices are treated as an array of column vectors with values taken from the input attributes identified in [Input attributes accessed by 32-bit input matrix variables](#). The `VkVertexInputAttributeDescription::format` must be specified with a `VkFormat` that corresponds to the appropriate type of column vector. The `Component` decoration must not be used with matrix types.

*Table 30. Input attributes accessed by 32-bit input matrix variables*

<b>Data type</b>	<b>Column vector type</b>	<b>Locations consumed</b>	<b>Components consumed</b>
mat2	two-component vector	i, i+1	(x, y, o, o), (x, y, o, o)
mat2x3	three-component vector	i, i+1	(x, y, z, o), (x, y, z, o)
mat2x4	four-component vector	i, i+1	(x, y, z, w), (x, y, z, w)
mat3x2	two-component vector	i, i+1, i+2	(x, y, o, o), (x, y, o, o), (x, y, o, o)
mat3	three-component vector	i, i+1, i+2	(x, y, z, o), (x, y, z, o), (x, y, z, o)
mat3x4	four-component vector	i, i+1, i+2	(x, y, z, w), (x, y, z, w), (x, y, z, w)
mat4x2	two-component vector	i, i+1, i+2, i+3	(x, y, o, o), (x, y, o, o), (x, y, o, o), (x, y, o, o)
mat4x3	three-component vector	i, i+1, i+2, i+3	(x, y, z, o), (x, y, z, o), (x, y, z, o), (x, y, z, o)
mat4	four-component vector	i, i+1, i+2, i+3	(x, y, z, w), (x, y, z, w), (x, y, z, w), (x, y, z, w)

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input (if present), or the default value.

When a vertex shader input variable declared using a scalar or vector 64-bit data type is assigned a location  $i$ , its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. The locations and components used depend on the type of variable and the `Component` decoration specified in the variable declaration, as identified in [Input attribute locations and components accessed by 64-bit input variables](#). For 64-bit data types, no default attribute values are provided. Input variables **must** not use more components than provided by the attribute. Input attributes which have one- or two-component 64-bit formats will consume a single location. Input attributes which have three- or four-component 64-bit formats will consume two consecutive locations. A 64-bit scalar data type will consume two components, and a 64-bit two-component vector data type will consume all four components available within a location. A three- or four-component 64-bit data type **must** not specify a component. A three-component 64-bit data type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations. A four-component 64-bit data type will consume all four components of the first location and all four components of the second location. It is invalid for a scalar or two-component 64-bit data type to specify a component of 1 or 3.

*Table 31. Input attribute locations and components accessed by 64-bit input variables*

Input format	Locations consumed	64-bit data type	Location decoration	Component decoration	32-bit components consumed
R64	$i$	scalar	$i$	0 or unspecified	(x, y, -, -)
R64G64	$i$	scalar	$i$	0 or unspecified	(x, y, o, o)
		scalar	$i$	2	(o, o, z, w)
		two-component vector	$i$	0 or unspecified	(x, y, z, w)
R64G64B64	$i, i+1$	scalar	$i$	0 or unspecified	(x, y, o, o), (o, o, -, -)
		scalar	$i$	2	(o, o, z, w), (o, o, -, -)
		scalar	$i+1$	0 or unspecified	(o, o, o, o), (x, y, -, -)
		two-component vector	$i$	0 or unspecified	(x, y, z, w), (o, o, -, -)
		three-component vector	$i$	unspecified	(x, y, z, w), (x, y, -, -)

Input format	Locations consumed	64-bit data type	Location decoration	Component decoration	32-bit components consumed
R64G64B64A64	i, i+1	scalar	i	0 or unspecified	(x, y, o, o), (o, o, o, o)
		scalar	i	2	(o, o, z, w), (o, o, o, o)
		scalar	i+1	0 or unspecified	(o, o, o, o), (x, y, o, o)
		scalar	i+1	2	(o, o, o, o), (o, o, z, w)
		two-component vector	i	0 or unspecified	(x, y, z, w), (o, o, o, o)
		two-component vector	i+1	0 or unspecified	(o, o, o, o), (x, y, z, w)
		three-component vector	i	unspecified	(x, y, z, w), (x, y, o, o)
		four-component vector	i	unspecified	(x, y, z, w), (x, y, z, w)

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute. Components indicated by “-” are not available for input variables as there are no default values provided for 64-bit data types, and there is no data provided by the input format.

When a vertex shader input variable declared using a 64-bit floating-point matrix type is assigned a location *i*, its values are taken from consecutive input attribute locations. Such matrices are treated as an array of column vectors with values taken from the input attributes as shown in [Input attribute locations and components accessed by 64-bit input variables](#). Each column vector starts at the location immediately following the last location of the previous column vector. The number of attributes and components assigned to each matrix is determined by the matrix dimensions and ranges from two to eight locations.

When a vertex shader input variable declared using an array type is assigned a location, its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. The number of attributes and components assigned to each element are determined according to the data type of the array elements and [Component decoration](#) (if any) specified in the declaration of the array, as described above. Each element of the array, in order, is assigned to consecutive locations, but all at the same specified component within each location.

Only input variables declared with the data types and component decorations as specified above are supported. *Location aliasing* is causing two variables to have the same location number. *Component aliasing* is assigning the same (or overlapping) component number for two location aliases. Location aliasing is allowed only if it does not cause component aliasing. Further, when location aliasing, the aliases sharing the location **must** all have the same SPIR-V floating-point component type or all have the same width integer-type components.

## 21.2. Vertex Input Description

Applications specify vertex input attribute and vertex input binding descriptions as part of graphics pipeline creation. `VkGraphicsPipelineCreateInfo::pVertexInputState` is a pointer to a `VkPipelineVertexInputStateCreateInfo` value.

The `VkPipelineVertexInputStateCreateInfo` structure is defined as:

```
typedef struct VkPipelineVertexInputStateCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineVertexInputStateCreateFlags   flags;
    uint32_t                  vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription* pVertexBindingDescriptions;
    uint32_t                  vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription* pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `vertexBindingDescriptionCount` is the number of vertex binding descriptions provided in `pVertexBindingDescriptions`.
- `pVertexBindingDescriptions` is a pointer to an array of `VkVertexInputBindingDescription` structures.
- `vertexAttributeDescriptionCount` is the number of vertex attribute descriptions provided in `pVertexAttributeDescriptions`.
- `pVertexAttributeDescriptions` is a pointer to an array of `VkVertexInputAttributeDescription` structures.

### Valid Usage

- `vertexBindingDescriptionCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`
- `vertexAttributeDescriptionCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputAttributes`
- For every `binding` specified by each element of `pVertexAttributeDescriptions`, a `VkVertexInputBindingDescription` **must** exist in `pVertexBindingDescriptions` with the same value of `binding`
- All elements of `pVertexBindingDescriptions` **must** describe distinct binding numbers
- All elements of `pVertexAttributeDescriptions` **must** describe distinct attribute locations

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkPipelineVertexInputDivisorStateCreateInfoEXT`
- `flags` **must** be `0`
- If `vertexBindingDescriptionCount` is not `0`, `pVertexBindingDescriptions` **must** be a valid pointer to an array of `vertexBindingDescriptionCount` valid `VkVertexInputBindingDescription` structures
- If `vertexAttributeDescriptionCount` is not `0`, `pVertexAttributeDescriptions` **must** be a valid pointer to an array of `vertexAttributeDescriptionCount` valid `VkVertexInputAttributeDescription` structures

```
typedef VkFlags VkPipelineVertexInputStateCreateFlags;
```

`VkPipelineVertexInputStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

Each vertex input binding is specified by an instance of the `VkVertexInputBindingDescription` structure.

The `VkVertexInputBindingDescription` structure is defined as:

```
typedef struct VkVertexInputBindingDescription {  
    uint32_t          binding;  
    uint32_t          stride;  
    VkVertexInputRate inputRate;  
} VkVertexInputBindingDescription;
```

- `binding` is the binding number that this structure describes.
- `stride` is the distance in bytes between two consecutive elements within the buffer.
- `inputRate` is a `VkVertexInputRate` value specifying whether vertex attribute addressing is a function of the vertex index or of the instance index.

## Valid Usage

- `binding` **must** be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`
- `stride` **must** be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindingStride`

## Valid Usage (Implicit)

- `inputRate` must be a valid `VkVertexInputRate` value

Possible values of `VkVertexInputBindingDescription::inputRate`, specifying the rate at which vertex attributes are pulled from buffers, are:

```
typedef enum VkVertexInputRate {
    VK_VERTEX_INPUT_RATE_VERTEX = 0,
    VK_VERTEX_INPUT_RATE_INSTANCE = 1,
    VK_VERTEX_INPUT_RATE_MAX_ENUM = 0x7FFFFFFF
} VkVertexInputRate;
```

- `VK_VERTEX_INPUT_RATE_VERTEX` specifies that vertex attribute addressing is a function of the vertex index.
- `VK_VERTEX_INPUT_RATE_INSTANCE` specifies that vertex attribute addressing is a function of the instance index.

Each vertex input attribute is specified by an instance of the `VkVertexInputAttributeDescription` structure.

The `VkVertexInputAttributeDescription` structure is defined as:

```
typedef struct VkVertexInputAttributeDescription {
    uint32_t    location;
    uint32_t    binding;
    VkFormat     format;
    uint32_t    offset;
} VkVertexInputAttributeDescription;
```

- `location` is the shader binding location number for this attribute.
- `binding` is the binding number which this attribute takes its data from.
- `format` is the size and type of the vertex attribute data.
- `offset` is a byte offset of this attribute relative to the start of an element in the vertex input binding.

## Valid Usage

- **location must** be less than `VkPhysicalDeviceLimits::maxVertexInputAttributes`
- **binding must** be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`
- **offset must** be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputAttributeOffset`
- **format must** be allowed as a vertex buffer format, as specified by the `VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT` flag in `VkFormatProperties::bufferFeatures` returned by `vkGetPhysicalDeviceFormatProperties`

## Valid Usage (Implicit)

- **format must** be a valid `VkFormat` value

To bind vertex buffers to a command buffer for use in subsequent draw commands, call:

```
void vkCmdBindVertexBuffers(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkBuffer*  
    const VkDeviceSize*  
                                commandBuffer,  
                                firstBinding,  
                                bindingCount,  
                                pBuffers,  
                                pOffsets);
```

- **commandBuffer** is the command buffer into which the command is recorded.
- **firstBinding** is the index of the first vertex input binding whose state is updated by the command.
- **bindingCount** is the number of vertex input bindings whose state is updated by the command.
- **pBuffers** is a pointer to an array of buffer handles.
- **pOffsets** is a pointer to an array of buffer offsets.

The values taken from elements *i* of **pBuffers** and **pOffsets** replace the current state for the vertex input binding **firstBinding** + *i*, for *i* in [0, **bindingCount**). The vertex input binding is updated to start at the offset indicated by **pOffsets[i]** from the start of the buffer **pBuffers[i]**. All vertex input attributes that use each of these bindings will use these updated addresses in their address calculations for subsequent draw commands.

## Valid Usage

- `firstBinding` **must** be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`
- The sum of `firstBinding` and `bindingCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`
- All elements of `pOffsets` **must** be less than the size of the corresponding element in `pBuffers`
- All elements of `pBuffers` **must** have been created with the `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` flag
- Each element of `pBuffers` that is non-sparse **must** be bound completely and contiguously to a single `VkDeviceMemory` object

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pBuffers` **must** be a valid pointer to an array of `bindingCount` valid `VkBuffer` handles
- `pOffsets` **must** be a valid pointer to an array of `bindingCount` `VkDeviceSize` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `bindingCount` **must** be greater than 0
- Both of `commandBuffer`, and the elements of `pBuffers` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

## 21.3. Vertex Attribute Divisor in Instanced Rendering

If `vertexAttributeInstanceRateDivisor` feature is enabled and the `pNext` chain of `VkPipelineVertexInputStateCreateInfo` includes a `VkPipelineVertexInputDivisorStateCreateInfoEXT` structure, then that structure controls how vertex attributes are assigned to an instance when instanced rendering is enabled.

The `VkPipelineVertexInputDivisorStateCreateInfoEXT` structure is defined as:

```
typedef struct VkPipelineVertexInputDivisorStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t vertexBindingDivisorCount;
    const VkVertexInputBindingDivisorDescriptionEXT* pVertexBindingDivisors;
} VkPipelineVertexInputDivisorStateCreateInfoEXT;
```

- `sType` is the type of this structure
- `pNext` is `NULL` or a pointer to an extension-specific structure
- `vertexBindingDivisorCount` is the number of elements in the `pVertexBindingDivisors` array.
- `pVertexBindingDivisors` is a pointer to an array of `VkVertexInputBindingDivisorDescriptionEXT` structures, which specifies the divisor value for each binding.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT`
- `pVertexBindingDivisors` **must** be a valid pointer to an array of `vertexBindingDivisorCount` `VkVertexInputBindingDivisorDescriptionEXT` structures
- `vertexBindingDivisorCount` **must** be greater than `0`

The individual divisor values per binding are specified using the `VkVertexInputBindingDivisorDescriptionEXT` structure which is defined as:

```
typedef struct VkVertexInputBindingDivisorDescriptionEXT {
    uint32_t binding;
    uint32_t divisor;
} VkVertexInputBindingDivisorDescriptionEXT;
```

- `binding` is the binding number for which the divisor is specified.
- `divisor` is the number of successive instances that will use the same value of the vertex attribute when instanced rendering is enabled. For example, if the divisor is `N`, the same vertex attribute will be applied to `N` successive instances before moving on to the next vertex attribute. The maximum value of divisor is implementation dependent and can be queried using `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor`. A value of `0` can

be used for the divisor if the `vertexAttributeInstanceRateZeroDivisor` feature is enabled. In this case, the same vertex attribute will be applied to all instances.

If this structure is not used to define a divisor value for an attribute then the divisor has a logical default value of 1.

## Valid Usage

- `binding` **must** be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`
- If the `vertexAttributeInstanceRateZeroDivisor` feature is not enabled, `divisor` **must** not be 0
- If the `vertexAttributeInstanceRateDivisor` feature is not enabled, `divisor` **must** be 1
- `divisor` **must** be a value between 0 and `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor`, inclusive.
- `VkVertexInputBindingDescription::inputRate` **must** be of type `VK_VERTEX_INPUT_RATE_INSTANCE` for this `binding`.

The address of each attribute for each `vertexIndex` and `instanceIndex` is calculated as follows:

- Let `attribDesc` be the member of `VkPipelineVertexInputStateCreateInfo ::pVertexAttributeDescriptions` with `VkVertexInputAttributeDescription::location` equal to the vertex input attribute number.
- Let `bindingDesc` be the member of `VkPipelineVertexInputStateCreateInfo ::pVertexBindingDescriptions` with `VkVertexInputAttributeDescription::binding` equal to `attribDesc.binding`.
- Let `vertexIndex` be the index of the vertex within the draw (a value between `firstVertex` and `firstVertex+vertexCount` for `vkCmdDraw`, or a value taken from the index buffer for `vkCmdDrawIndexed`), and let `instanceIndex` be the instance number of the draw (a value between `firstInstance` and `firstInstance+instanceCount`).
- Let `divisor` be the member of `VkPipelineVertexInputDivisorStateCreateInfoEXT ::pVertexBindingDivisors` with `VkVertexInputBindingDivisorDescriptionEXT::binding` equal to `attribDesc.binding`.

```

bufferBindingAddress = buffer[binding].baseAddress + offset[binding];

if (bindingDesc.inputRate == VK_VERTEX_INPUT_RATE_VERTEX)
    vertexOffset = vertexIndex * bindingDesc.stride;
else
    if (divisor == 0)
        vertexOffset = firstInstance * bindingDesc.stride;
    else
        vertexOffset = (firstInstance + ((instanceIndex - firstInstance) / divisor)) *
bindingDesc.stride;

attribAddress = bufferBindingAddress + vertexOffset + attribDesc.offset;

```

For each attribute, raw data is extracted starting at `attribAddress` and is converted from the `VkVertexInputAttributeDescription`'s `format` to either to floating-point, unsigned integer, or signed integer based on the base type of the format; the base type of the format **must** match the base type of the input variable in the shader. If `format` is a packed format, `attribAddress` **must** be a multiple of the size in bytes of the whole attribute data type as described in [Packed Formats](#). Otherwise, `attribAddress` **must** be a multiple of the size in bytes of the component type indicated by `format` (see [Formats](#)). If the format does not include G, B, or A components, then those are filled with (0,0,1) as needed (using either 1.0f or integer 1 based on the format) for attributes that are not 64-bit data types. The number of components in the vertex shader input variable need not exactly match the number of components in the format. If the vertex shader has fewer components, the extra components are discarded.

## 21.4. Example

To create a graphics pipeline that uses the following vertex description:

```

struct Vertex
{
    float    x, y, z, w;
    uint8_t u, v;
};

```

The application could use the following set of structures:

```

const VkVertexInputBindingDescription binding =
{
    0,                                // binding
    sizeof(Vertex),                  // stride
    VK_VERTEX_INPUT_RATE_VERTEX        // inputRate
};

const VkVertexInputAttributeDescription attributes[] =
{
{
    0,                                // location
    binding.binding,                  // binding
    VK_FORMAT_R32G32B32A32_SFLOAT,    // format
    0                                 // offset
},
{
    1,                                // location
    binding.binding,                  // binding
    VK_FORMAT_R8G8_UNORM,             // format
    4 * sizeof(float)                // offset
}
};

const VkPipelineVertexInputStateCreateInfo viInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_CREATE_INFO, // sType
    NULL,                               // pNext
    0,                                 // flags
    1,                                 // vertexBindingDescriptionCount
    &binding,                           // pVertexBindingDescriptions
    2,                                 // vertexAttributeDescriptionCount
    &attributes[0]                      // pVertexAttributeDescriptions
};

```

# Chapter 22. Tessellation

Tessellation involves three pipeline stages. First, a [tessellation control shader](#) transforms control points of a patch and **can** produce per-patch data. Second, a fixed-function tessellator generates multiple primitives corresponding to a tessellation of the patch in (u,v) or (u,v,w) parameter space. Third, a [tessellation evaluation shader](#) transforms the vertices of the tessellated patch, for example to compute their positions and attributes as part of the tessellated surface. The tessellator is enabled when the pipeline contains both a tessellation control shader and a tessellation evaluation shader.

## 22.1. Tessellator

If a pipeline includes both tessellation shaders (control and evaluation), the tessellator consumes each input patch (after vertex shading) and produces a new set of independent primitives (points, lines, or triangles). These primitives are logically produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch outer and inner tessellation levels written by the tessellation control shader. These levels are specified using the [built-in variables](#) `TessLevelOuter` and `TessLevelInner`, respectively. This subdivision is performed in an implementation-dependent manner. If no tessellation shaders are present in the pipeline, the tessellator is disabled and incoming primitives are passed through without modification.

The type of subdivision performed by the tessellator is specified by an `OpExecutionMode` instruction in the tessellation evaluation or tessellation control shader using one of execution modes `Triangles`, `Quads`, and `Isolines`. Other tessellation-related execution modes **can** also be specified in either the tessellation control or tessellation evaluation shaders, and if they are specified in both then the modes **must** be the same.

Tessellation execution modes include:

- `Triangles`, `Quads`, and `Isolines`. These control the type of subdivision and topology of the output primitives. One mode **must** be set in at least one of the tessellation shader stages.
- `VertexOrderCw` and `VertexOrderCcw`. These control the orientation of triangles generated by the tessellator. One mode **must** be set in at least one of the tessellation shader stages.
- `PointMode`. Controls generation of points rather than triangles or lines. This functionality defaults to disabled, and is enabled if either shader stage includes the execution mode.
- `SpacingEqual`, `SpacingFractionalEven`, and `SpacingFractionalOdd`. Controls the spacing of segments on the edges of tessellated primitives. One mode **must** be set in at least one of the tessellation shader stages.
- `OutputVertices`. Controls the size of the output patch of the tessellation control shader. One value **must** be set in at least one of the tessellation shader stages.

For triangles, the tessellator subdivides a triangle primitive into smaller triangles. For quads, the tessellator subdivides a rectangle primitive into smaller triangles. For isolines, the tessellator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching across the rectangle in the u dimension (i.e. the coordinates in `TessCoord` are of the form (0,x) through (1,x) for all tessellation evaluation shader invocations that share a line).

Each vertex produced by the tessellator has an associated (u,v,w) or (u,v) position in a normalized parameter space, with parameter values in the range [0,1], as illustrated in figures [Domain parameterization for tessellation primitive modes \(upper-left origin\)](#) and [Domain parameterization for tessellation primitive modes \(lower-left origin\)](#). The domain space **can** have either an upper-left or lower-left origin, selected by the `domainOrigin` member of `VkPipelineTessellationDomainOriginStateCreateInfo`.

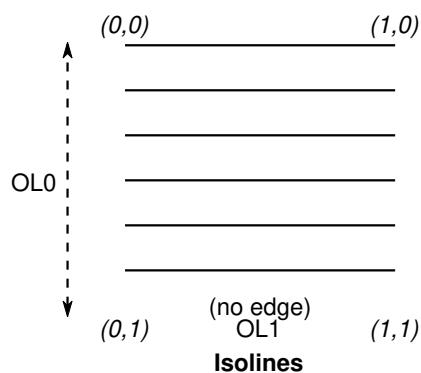
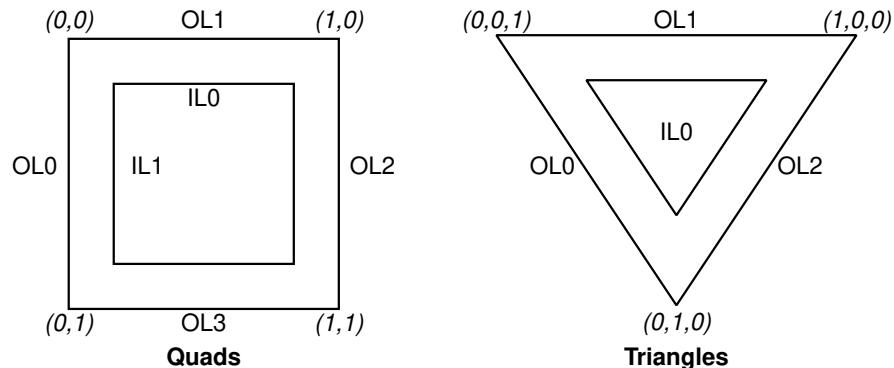


Figure 13. Domain parameterization for tessellation primitive modes (upper-left origin)

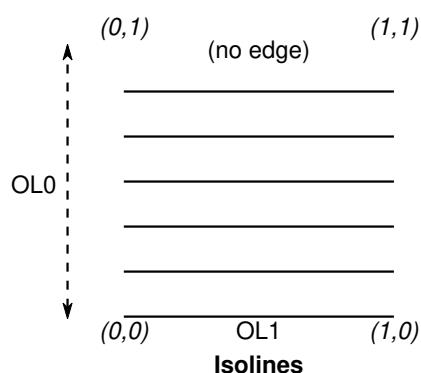
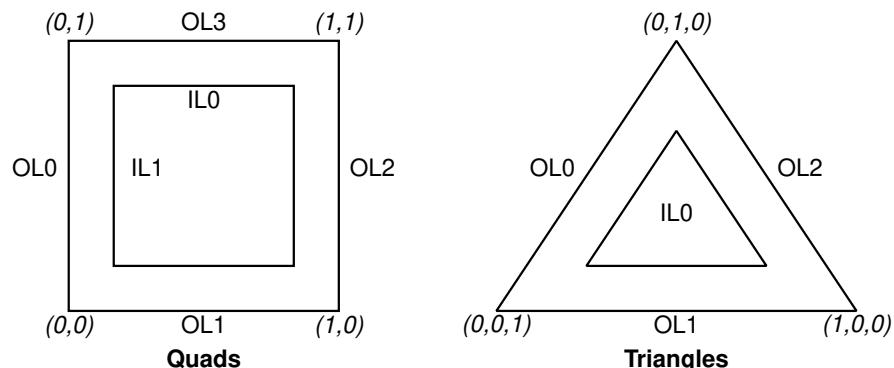


Figure 14. Domain parameterization for tessellation primitive modes (lower-left origin)

## Caption

In the domain parameterization diagrams, the coordinates illustrate the value of `TessCoord` at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

For triangles, the vertex's position is a barycentric coordinate (u,v,w), where  $u + v + w = 1.0$ , and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a (u,v) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

## 22.2. Tessellator Patch Discard

A patch is discarded by the tessellator if any relevant outer tessellation level is less than or equal to zero.

Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point NaN (not a number) in implementations supporting NaN.

No new primitives are generated and the tessellation evaluation shader is not executed for patches that are discarded. For `Quads`, all four outer levels are relevant. For `Triangles` and `IsoLines`, only the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.

## 22.3. Tessellator Spacing

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an `OpExecutionMode` in the tessellation control or tessellation evaluation shader using one of the identifiers `SpacingEqual`, `SpacingFractionalEven`, or `SpacingFractionalOdd`.

If `SpacingEqual` is used, the floating-point tessellation level is first clamped to  $[1, \text{maxLevel}]$ , where `maxLevel` is the implementation-dependent maximum tessellation level (`VkPhysicalDeviceLimits ::maxTessellationGenerationLevel`). The result is rounded up to the nearest integer n, and the corresponding edge is divided into n segments of equal length in (u,v) space.

If `SpacingFractionalEven` is used, the tessellation level is first clamped to  $[2, \text{maxLevel}]$  and then rounded up to the nearest even integer n. If `SpacingFractionalOdd` is used, the tessellation level is clamped to  $[1, \text{maxLevel} - 1]$  and then rounded up to the nearest odd integer n. If n is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into  $n - 2$  segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with  $n - f$ , where f is the clamped floating-point tessellation level. When  $n - f$  is zero, the additional segments will have equal length to the other segments. As  $n - f$  approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments **must** be

placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is implementation-dependent, but **must** be identical for any pair of subdivided edges with identical values of  $f$ .

When tessellating triangles or quads using [point mode](#) with fractional odd spacing, the tessellator **may** produce *interior vertices* that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

## 22.4. Tessellation Primitive Ordering

Few guarantees are provided for the relative ordering of primitives produced by tessellation, as they pertain to [primitive order](#).

- The output primitives generated from each input primitive are passed to subsequent pipeline stages in an implementation-dependent order.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

## 22.5. Tessellator Vertex Winding Order

When the tessellator produces triangles (in the [Triangles](#) or [Quads](#) modes), the orientation of all triangles is specified with an [OpExecutionMode](#) of [VertexOrderCw](#) or [VertexOrderCcw](#) in the tessellation control or tessellation evaluation shaders. If the order is [VertexOrderCw](#), the vertices of all generated triangles will have clockwise ordering in  $(u,v)$  or  $(u,v,w)$  space. If the order is [VertexOrderCcw](#), the vertices will have counter-clockwise ordering in that space.

If the tessellation domain has an upper-left origin, the vertices of a triangle have counter-clockwise ordering if

$$a = u_0 v_1 - u_1 v_0 + u_1 v_2 - u_2 v_1 + u_2 v_0 - u_0 v_2$$

is negative, and clockwise ordering if  $a$  is positive.  $u_i$  and  $v_i$  are the  $u$  and  $v$  coordinates in normalized parameter space of the  $i$ th vertex of the triangle. If the tessellation domain has a lower-left origin, the vertices of a triangle have counter-clockwise ordering if  $a$  is positive, and clockwise ordering if  $a$  is negative.

### Note

The value  $a$  is proportional (with a positive factor) to the signed area of the triangle.



In [Triangles](#) mode, even though the vertex coordinates have a  $w$  value, it does not participate directly in the computation of  $a$ , being an affine combination of  $u$  and  $v$ .

## 22.6. Triangle Tessellation

If the tessellation primitive mode is **Triangles**, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the  $u = 0$  (left),  $v = 0$  (bottom), and  $w = 0$  (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with  $(u,v,w)$  coordinates of  $(0,0,1)$ ,  $(1,0,0)$ , and  $(0,1,0)$  is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as  $1 + \epsilon$  and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision **may** produce *inner vertices* positioned on the edge of the triangle.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating  $n$  segments. For the outermost inner triangle, the inner triangle is degenerate—a single point at the center of the triangle—if  $n$  is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the corner's two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If  $n$  is three, the edges of the inner triangle are not subdivided and is the final triangle in the set of concentric triangles. Otherwise, each edge of the inner triangle is divided into  $n - 2$  segments, with the  $n - 1$  vertices of this subdivision produced by intersecting the inner edge with lines perpendicular to the edge running through the  $n - 1$  innermost vertices of the subdivision of the outer edge. Once the outermost inner triangle is subdivided, the previous subdivision process repeats itself, using the generated triangle as an outer triangle. This subdivision process is illustrated in [Inner Triangle Tessellation](#).

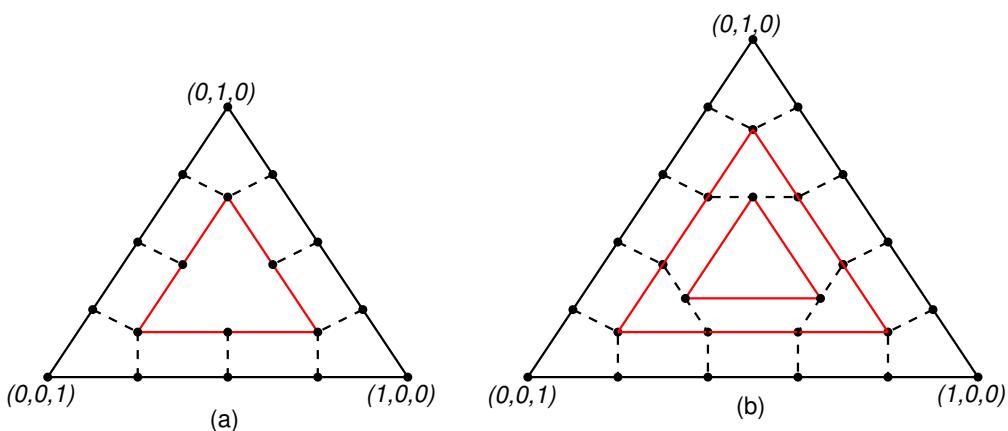


Figure 15. Inner Triangle Tessellation

## Caption

In the [Inner Triangle Tessellation](#) diagram, inner tessellation levels of (a) five and (b) four are shown (not to scale). Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge passing through each inner vertex of the subdivided outer edge. Dotted lines depict edges connecting corresponding vertices on the inner and outer triangle edges.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.

After the area corresponding to any inner triangles is filled, the tessellator generates triangles to cover the area between the outermost triangle and the outermost inner triangle. To do this, the temporary subdivision of the outer triangle edge above is discarded. Instead, the  $u = 0$ ,  $v = 0$ , and  $w = 0$  edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point.

After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric  $(u,v,w)$  coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in  $(u,v,w)$  space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles.

Output triangles are generated with a topology similar to [triangle lists](#), except that the order in which each triangle is generated, and the order in which the vertices are generated for each triangle, are implementation-dependent. However, the order of vertices in each triangle is consistent across the domain as described in [Tessellator Vertex Winding Order](#).

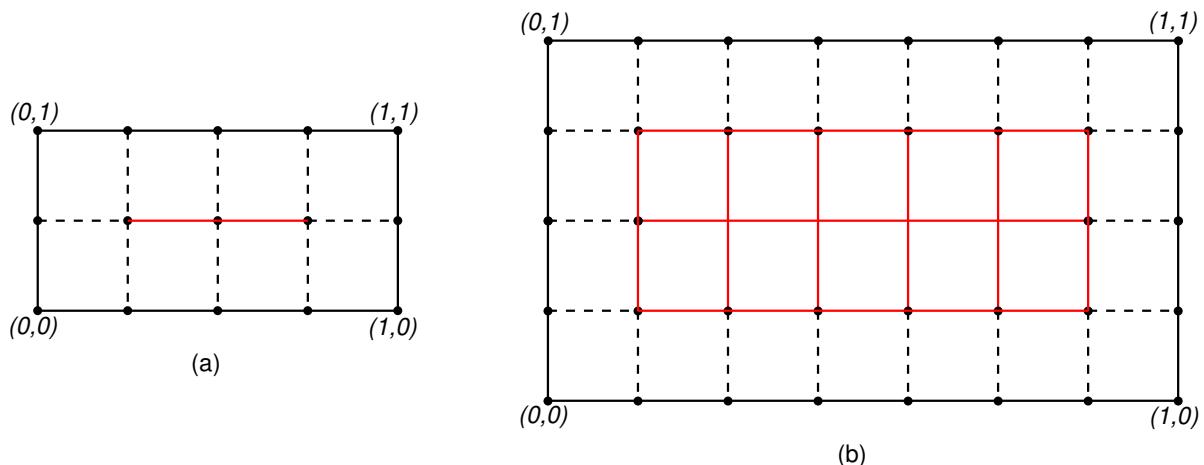
## 22.7. Quad Tessellation

If the tessellation primitive mode is [Quads](#), a rectangle is subdivided into a collection of triangles covering the area of the original rectangle. First, the original rectangle is subdivided into a regular mesh of rectangles, where the number of rectangles along the  $u = 0$  and  $u = 1$  (vertical) and  $v = 0$  and  $v = 1$  (horizontal) edges are derived from the first and second inner tessellation levels, respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using

the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of the  $u = 0$  (left),  $v = 0$  (bottom),  $u = 1$  (right), and  $v = 1$  (top) edges, respectively. The area between the inner rectangles of the mesh and the outer rectangle edges are filled by triangles produced by joining the vertices on the subdivided outer edges to the vertices on the edge of the inner rectangle mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either clamped inner tessellation level is one, that tessellation level is treated as though it were originally specified as  $1 + \varepsilon$  and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision **may** produce *inner vertices* positioned on the edge of the rectangle.

If any tessellation level is greater than one, tessellation begins by subdividing the  $u = 0$  and  $u = 1$  edges of the outer rectangle into  $m$  segments using the clamped and rounded first inner tessellation level and the tessellation spacing. The  $v = 0$  and  $v = 1$  edges are subdivided into  $n$  segments using the second inner tessellation level. Each vertex on the  $u = 0$  and  $v = 0$  edges are joined with the corresponding vertex on the  $u = 1$  and  $v = 1$  edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either  $m$  or  $n$  is two, the inner rectangle is degenerate, and one or both of the rectangle's *edges* consist of a single point. This subdivision is illustrated in Figure [Inner Quad Tessellation](#).



*Figure 16. Inner Quad Tessellation*

### Caption

In the [Inner Quad Tessellation](#) diagram, inner quad tessellation levels of (a) (4,2) and (b) (7,4) are shown. The regions highlighted in red in figure (b) depict the 10 inner rectangles, each of which will be subdivided into two triangles. Solid black circles depict vertices on the boundary of the outer and inner rectangles, where the inner rectangle on the top figure is degenerate (a single line segment). Dotted lines depict the horizontal and vertical edges connecting corresponding vertices on the inner and outer rectangle edges.

After the area corresponding to the inner rectangle is filled, the tessellator **must** produce triangles to cover the area between the inner and outer rectangles. To do this, the subdivision of the outer rectangle edge above is discarded. Instead, the  $u = 0$ ,  $v = 0$ ,  $u = 1$ , and  $v = 1$  edges are subdivided according to the first, second, third, and fourth outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the inner rectangle is retained. The area between the outer and inner rectangles is completely filled by non-overlapping triangles. Two of the three vertices of each triangle are adjacent vertices on a subdivided edge of one rectangle; the third is one of the vertices on the corresponding edge of the other rectangle. If either edge of the innermost rectangle is degenerate, the area near the corresponding outer edges is filled by connecting each vertex on the outer edge with the single vertex making up the *inner edge*.

The algorithm used to subdivide the rectangular domain in  $(u,v)$  space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles.

Output triangles are generated with a topology similar to [triangle lists](#), except that the order in which each triangle is generated, and the order in which the vertices are generated for each triangle, are implementation-dependent. However, the order of vertices in each triangle is consistent across the domain as described in [Tessellator Vertex Winding Order](#).

## 22.8. Isoline Tessellation

If the tessellation primitive mode is [Isolines](#), a set of independent horizontal line segments is drawn. The segments are arranged into connected strips called *isolines*, where the vertices of each isoline have a constant  $v$  coordinate and  $u$  coordinates covering the full range  $[0,1]$ . The number of isolines generated is derived from the first outer tessellation level; the number of segments in each isoline is derived from the second outer tessellation level. Both inner tessellation levels and the third and fourth outer tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. The  $u = 0$  and  $u = 1$  edges of the rectangle are subdivided according to the first outer tessellation level. For the purposes of this subdivision, the tessellation spacing mode is ignored and treated as `equal_spacing`. An isoline is drawn connecting each vertex on the  $u = 0$  rectangle edge to the corresponding vertex on the  $u = 1$  rectangle edge, except that no line is drawn between  $(0,1)$  and  $(1,1)$ . If the number of isolines on the subdivided  $u = 0$  and  $u = 1$  edges is  $n$ , this process will result in  $n$  equally spaced lines with constant  $v$  coordinates of  $0, \frac{1}{n}, \frac{2}{n}, \dots, \frac{n-1}{n}$ .

Each of the  $n$  isolines is then subdivided according to the second outer tessellation level and the tessellation spacing, resulting in  $m$  line segments. Each segment of each line is emitted by the tessellator. These line segments are generated with a topology similar to [line lists](#), except that the order in which each line is generated, and the order in which the vertices are generated for each line segment, are implementation-dependent.

## 22.9. Tessellation Point Mode

For all primitive modes, the tessellator is capable of generating points instead of lines or triangles. If the tessellation control or tessellation evaluation shader specifies the [OpExecutionMode PointMode](#), the primitive generator will generate one point for each distinct vertex produced by tessellation,

rather than emitting triangles or lines. Otherwise, the tessellator will produce a collection of line segments or triangles according to the primitive mode. These points are generated with a topology similar to [point lists](#), except the order in which the points are generated for each input primitive is undefined.

## 22.10. Tessellation Pipeline State

The `pTessellationState` member of [VkGraphicsPipelineCreateInfo](#) is a pointer to a [VkPipelineTessellationStateCreateInfo](#) structure.

The [VkPipelineTessellationStateCreateInfo](#) structure is defined as:

```
typedef struct VkPipelineTessellationStateCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineTessellationStateCreateFlags   flags;
    uint32_t                  patchControlPoints;
} VkPipelineTessellationStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `patchControlPoints` number of control points per patch.

### Valid Usage

- `patchControlPoints` **must** be greater than zero and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of [VkPipelineTessellationDomainOriginStateCreateInfo](#)
- `flags` **must** be `0`

```
typedef VkFlags VkPipelineTessellationStateCreateFlags;
```

`VkPipelineTessellationStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The [VkPipelineTessellationDomainOriginStateCreateInfo](#) structure is defined as:

```
typedef struct VkPipelineTessellationDomainOriginStateCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkTessellationDomainOrigin domainOrigin;
} VkPipelineTessellationDomainOriginStateCreateInfo;
```

or the equivalent

```
typedef VkPipelineTessellationDomainOriginStateCreateInfo
VkPipelineTessellationDomainOriginStateCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `domainOrigin` is a `VkTessellationDomainOrigin` value controlling the origin of the tessellation domain space.

If the `VkPipelineTessellationDomainOriginStateCreateInfo` structure is included in the `pNext` chain of `VkPipelineTessellationStateCreateInfo`, it controls the origin of the tessellation domain. If this structure is not present, it is as if `domainOrigin` were `VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO`
- `domainOrigin` **must** be a valid `VkTessellationDomainOrigin` value

The possible tessellation domain origins are specified by the `VkTessellationDomainOrigin` enumeration:

```
typedef enum VkTessellationDomainOrigin {
    VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT = 0,
    VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT = 1,
    VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT_KHR =
VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT,
    VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT_KHR =
VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT,
    VK_TESSELLATION_DOMAIN_ORIGIN_MAX_ENUM = 0x7FFFFFFF
} VkTessellationDomainOrigin;
```

or the equivalent

```
typedef VkTessellationDomainOrigin VkTessellationDomainOriginKHR;
```

- `VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT` specifies that the origin of the domain space is in the

upper left corner, as shown in figure [Domain parameterization for tessellation primitive modes \(upper-left origin\)](#).

- `VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT` specifies that the origin of the domain space is in the lower left corner, as shown in figure [Domain parameterization for tessellation primitive modes \(lower-left origin\)](#).

This enum affects how the `VertexOrderCw` and `VertexOrderCcw` tessellation execution modes are interpreted, since the winding is defined relative to the orientation of the domain.

# Chapter 23. Geometry Shading

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive. Geometry shading is enabled when a geometry shader is included in the pipeline.

## 23.1. Geometry Shader Input Primitives

Each geometry shader invocation has access to all vertices in the primitive (and their associated data), which are presented to the shader as an array of inputs.

The input primitive type expected by the geometry shader is specified with an `OpExecutionMode` instruction in the geometry shader, and **must** match the incoming primitive type specified by either the pipeline's `primitive topology` if tessellation is inactive, or the `tessellation mode` if tessellation is active, as follows:

- An input primitive type of `InputPoints` **must** only be used with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, or with a tessellation shader that specifies `PointMode`. The input arrays always contain one element, as described by the `point list topology` or `tessellation in point mode`.
- An input primitive type of `InputLines` **must** only be used with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` or `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`, or with a tessellation shader specifying `Isolines` that does not specify `PointMode`. The input arrays always contain two elements, as described by the `line list topology` or `line strip topology`, or by `isoline tessellation`.
- An input primitive type of `InputLinesAdjacency` **must** only be used when tessellation is inactive, with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`. The input arrays always contain four elements, as described by the `line list with adjacency topology` or `line strip with adjacency topology`.
- An input primitive type of `Triangles` **must** only be used with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`; or with a tessellation shader specifying `Quads` or `Triangles` that does not specify `PointMode`. The input arrays always contain three elements, as described by the `triangle list topology`, `triangle strip topology`, or `triangle fan topology`, or by `triangle` or `quad tessellation`. Vertices **may** be in a different absolute order to that specified by the topology, but **must** adhere to the specified winding order.
- An input primitive type of `InputTrianglesAdjacency` **must** only be used when tessellation is inactive, with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`. The input arrays always contain six elements, as described by the `triangle list with adjacency topology` or `triangle strip with adjacency topology`. Vertices **may** be in a different absolute order to that specified by the topology, but **must** adhere to the specified winding order, and the vertices making up the main primitive **must** still occur at the first, third, and fifth index.

## 23.2. Geometry Shader Output Primitives

A geometry shader generates primitives in one of three output modes: points, line strips, or triangle strips. The primitive mode is specified in the shader using an `OpExecutionMode` instruction with the `OutputPoints`, `OutputLineStrip` or `OutputTriangleStrip` modes, respectively. Each geometry shader **must** include exactly one output primitive mode.

The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type and the resulting primitives are then further processed as described in [Rasterization](#). If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, vertices corresponding to incomplete primitives are not processed by subsequent pipeline stages. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The maximum output vertex count is specified in the shader using an `OpExecutionMode` instruction with the mode set to `OutputVertices` and the maximum number of vertices that will be produced by the geometry shader specified as a literal. Each geometry shader **must** specify a maximum output vertex count.

## 23.3. Multiple Invocations of Geometry Shaders

Geometry shaders **can** be invoked more than one time for each input primitive. This is known as *geometry shader instancing* and is requested by including an `OpExecutionMode` instruction with `mode` specified as `Invocations` and the number of invocations specified as an integer literal.

In this mode, the geometry shader will execute at least n times for each input primitive, where n is the number of invocations specified in the `OpExecutionMode` instruction. The instance number is available to each invocation as a built-in input using `InvocationId`.

## 23.4. Geometry Shader Primitive Ordering

Limited guarantees are provided for the relative ordering of primitives produced by a geometry shader, as they pertain to [primitive order](#).

- For instanced geometry shaders, the output primitives generated from each input primitive are passed to subsequent pipeline stages using the invocation number to order the primitives, from least to greatest.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

## 23.5. Geometry Shader Passthrough

A geometry shader that uses the `PassthroughNV` decoration on a variable in its input interface is considered a *passthrough geometry shader*. Output primitives in a passthrough geometry shader **must** have the same topology as the input primitive and are not produced by emitting vertices. The vertices of the output primitive have two different types of attributes, per-vertex and per-primitive. Geometry shader input variables with `PassthroughNV` decoration are considered to produce per-

vertex outputs, where values for each output vertex are copied from the corresponding input vertex. Any built-in or user-defined geometry shader outputs are considered per-primitive in a passthrough geometry shader, where a single output value is copied to all output vertices.

The remainder of this section details the usage of the `PassthroughNV` decoration and modifications to the interface matching rules when using passthrough geometry shaders.

### 23.5.1. `PassthroughNV` Decoration

Decorating a geometry shader input variable with the `PassthroughNV` decoration indicates that values of this input are copied through to the corresponding vertex of the output primitive. Input variables and block members which do not have the `PassthroughNV` decoration are consumed by the geometry shader without being passed through to subsequent stages.

The `PassthroughNV` decoration **must** only be used within a geometry shader.

Any variable decorated with `PassthroughNV` **must** be declared using the `Input` storage class.

The `PassthroughNV` decoration **must** not be used with any of:

- an input primitive type other than `InputPoints`, `InputLines`, or `Triangles`, as specified by the mode for `OpExecutionMode`.
- an invocation count other than one, as specified by the `Invocations` mode for `OpExecutionMode`.
- an `OpEntryPoint` which statically uses the `OpEmitVertex` or `OpEndPrimitive` instructions.
- a variable decorated with the `InvocationId` built-in decoration.
- a variable decorated with the `PrimitiveId` built-in decoration that is declared using the `Input` storage class.

### 23.5.2. Passthrough Interface Matching

When a passthrough geometry shader is in use, the `Interface Matching` rules involving the geometry shader input and output interfaces operate as described in this section.

For the purposes of matching passthrough geometry shader inputs with outputs of the previous pipeline stages, the `PassthroughNV` decoration is ignored.

For the purposes of matching the outputs of the geometry shader with subsequent pipeline stages, each input variable with the `PassthroughNV` decoration is considered to add an equivalent output variable with the same type, decoration (other than `PassthroughNV`), number, and declaration order on the output interface. The output variable declaration corresponding to an input variable decorated with `PassthroughNV` will be identical to the input declaration, except that the outermost array dimension of such variables is removed. The output block declaration corresponding to an input block decorated with `PassthroughNV` or having members decorated with `PassthroughNV` will be identical to the input declaration, except that the outermost array dimension of such declaration is removed.

If an input block is decorated with `PassthroughNV`, the equivalent output block contains all the members of the input block. Otherwise, the equivalent output block contains only those input block

members decorated with `PassthroughNV`. All members of the corresponding output block are assigned `Location` and `Component` decorations identical to those assigned to the corresponding input block members.

Output variables and blocks generated from inputs decorated with `PassthroughNV` will only exist for the purposes of interface matching; these declarations are not available to geometry shader code or listed in the module interface.

For the purposes of component counting, passthrough geometry shaders count all statically used input variable components declared with the `PassthroughNV` decoration as output components as well, since their values will be copied to the output primitive produced by the geometry shader.

# Chapter 24. Mesh Shading

[Task](#) and [mesh shaders](#) operate in workgroups to produce a collection of primitives that will be processed by subsequent stages of the graphics pipeline.

Work on the mesh pipeline is initiated by the application [drawing](#) a set of mesh tasks organized in global workgroups. If the optional task shader is active, each workgroup triggers the execution of task shader invocations that will create a new set of mesh workgroups upon completion. Each of these created workgroups, or each of the original workgroups if no task shader is present, triggers the execution of mesh shader invocations.

Each mesh shader workgroup emits zero or more output primitives along with the group of vertices and their associated data required for each output primitive.

## 24.1. Task Shader Input

For every workgroup issued via the drawing commands a group of task shader invocations is executed. There are no inputs other than the builtin workgroup identifiers.

## 24.2. Task Shader Output

The task shader can emit zero or more mesh workgroups to be generated using the [built-in variable TaskCountNV](#). This value **must** be less than or equal to [VkPhysicalDeviceMeshShaderPropertiesNV::maxTaskOutputCount](#).

It can also output user-defined data that is passed as input to all mesh shader invocations that the task creates. These outputs are decorated as [PerTaskNV](#).

## 24.3. Mesh Generation

If a task shader exists, the mesh assembler creates a variable amount of mesh workgroups depending on each task's output. If there is no task shader, the drawing commands emit the mesh shader invocations directly.

## 24.4. Mesh Shader Input

The only inputs available to the mesh shader are variables identifying the specific workgroup and invocation and, if applicable, any outputs written as [PerTaskNV](#) by the task shader that spawned the mesh shader's workgroup. The mesh shader can operate without a task shader as well.

## 24.5. Mesh Shader Output Primitives

A mesh shader generates primitives in one of three output modes: points, lines, or triangles. The primitive mode is specified in the shader using an [OpExecutionMode](#) instruction with the [OutputPoints](#), [OutputLinesNV](#), or [OutputTrianglesNV](#) modes, respectively. Each mesh shader **must** include exactly one output primitive mode.

The maximum output vertex count is specified as a literal in the shader using an `OpExecutionMode` instruction with the mode set to `OutputVertices` and **must** be less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputVertices`.

The maximum output primitive count is specified as a literal in the shader using an `OpExecutionMode` instruction with the mode set to `OutputPrimitivesNV` and **must** be less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputPrimitives`.

The number of primitives output by the mesh shader is provided via writing to the [built-in variable](#) `PrimitiveCountNV` and **must** be less than or equal to the maximum output primitive count specified in the shader. A variable decorated with `PrimitiveIndicesNV` is an output array of local index values into the vertex output arrays from which primitives are assembled according to the output primitive type. These resulting primitives are then further processed as described in [Rasterization](#).

## 24.6. Mesh Shader Per-View Outputs

The mesh shader outputs decorated with the `PositionPerViewNV`, `ClipDistancePerViewNV`, `CullDistancePerViewNV`, `LayerPerViewNV`, and `ViewportMaskPerViewNV` built-in decorations are the per-view versions of the single-view variables with equivalent names (that is `Position`, `ClipDistance`, `CullDistance`, `Layer`, and `ViewportMaskNV`, respectively). If a shader statically assigns a value to any element of a per-view array it **must** not statically assign a value to the equivalent single-view variable.

Each of these outputs is considered arrayed, with separate values for each view. The view number is used to index the first dimension of these arrays.

The second dimension of the `ClipDistancePerViewNV`, and `CullDistancePerViewNV` arrays have the same requirements as the `ClipDistance`, and `CullDistance` arrays.

If a mesh shader output is *per-view*, the corresponding fragment shader input is taken from the element of the per-view output array that corresponds to the view that is currently being processed by the fragment shader.

## 24.7. Mesh Shader Primitive Ordering

Following guarantees are provided for the relative ordering of primitives produced by a mesh shader, as they pertain to [primitive order](#).

- When a task shader is used, mesh workgroups spawned from lower tasks will be ordered prior those workgroups from subsequent tasks.
- All output primitives generated from a given mesh workgroup are passed to subsequent pipeline stages before any output primitives generated from subsequent input workgroups.
- All output primitives within a mesh workgroup, will be generated in the ordering provided by the builtin primitive indexbuffer (from low address to high address).

# Chapter 25. Fixed-Function Vertex Post-Processing

After programmable vertex processing, the following fixed-function operations are applied to vertices of the resulting primitives:

- Transform feedback (see [Transform Feedback](#))
- Viewport swizzle (see [Viewport Swizzle](#))
- Flat shading (see [Flat Shading](#)).
- Primitive clipping, including client-defined half-spaces (see [Primitive Clipping](#)).
- Shader output attribute clipping (see [Clipping Shader Outputs](#)).
- Clip space W scaling (see [Controlling Viewport W Scaling](#)).
- Perspective division on clip coordinates (see [Coordinate Transformations](#)).
- Viewport mapping, including depth range scaling (see [Controlling the Viewport](#)).
- Front face determination for polygon primitives (see [Basic Polygon Rasterization](#)).

Next, rasterization is performed on primitives as described in chapter [Rasterization](#).

## 25.1. Transform Feedback

Before any other fixed-function vertex post-processing, vertex outputs from the last shader in the vertex processing stage **can** be written out to one or more transform feedback buffers bound to the command buffer. To capture vertex outputs the last vertex processing stage shader **must** be declared with the `Xfb` execution mode. Outputs decorated with `XfbBuffer` will be written out to the corresponding transform feedback buffers bound to the command buffer when transform feedback is active. Transform feedback buffers are bound to the command buffer by using `vkCmdBindTransformFeedbackBuffersEXT`. Transform feedback is made active by calling `vkCmdBeginTransformFeedbackEXT` and made inactive by calling `vkCmdEndTransformFeedbackEXT`. After vertex data is written it is possible to use `vkCmdDrawIndirectByteCountEXT` to start a new draw where the `vertexCount` is derived from the number of bytes written by a previous transform feedback.

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active, the values of the specified output variables are assembled into primitives and appended to the bound transform feedback buffers. After activating transform feedback, the values of the first assembled primitive are written at the starting offsets of the bound transform feedback buffers, and subsequent primitives are appended to the buffer. If the optional `pCounterBuffers` and `pCounterBufferOffsets` parameters are specified, the starting points within the transform feedback buffers are adjusted so data is appended to the previously written values indicated by the value stored by the implementation in the counter buffer.

For multi-vertex primitives, all values for a given vertex are written before writing values for any other vertex. Implementations **may** write out any vertex within the primitive first, but all subsequent vertices for that primitive **must** be written out in a consistent winding order defined as

follows:

- If neither [geometry](#) or [tessellation shading](#) is active, vertices within a primitive are appended according to the winding order described by the [primitive topology](#) defined by the [VkPipelineInputAssemblyStateCreateInfo:topology](#) used to execute the [drawing command](#).
- If [geometry shading](#) is active, vertices within a primitive are appended according to the winding order described by the [primitive topology](#) defined by the [OutputPoints](#), [OutputLineStrips](#), or [OutputTriangleStrips](#) execution mode.
- If [tessellation shading](#) is active but [geometry shading](#) is not, vertices within a primitive are appended according to the winding order defined by [triangle tessellation](#), [quad tessellation](#), and [isoline tessellation](#).

When capturing vertices, the stride associated with each transform feedback buffer, as indicated by the [XfbStride](#) decoration, indicates the number of bytes of storage reserved for each vertex in the transform feedback buffer. For every vertex captured, each output attribute with a [Offset](#) decoration will be written to the storage reserved for the vertex at the associated transform feedback buffer. When writing output variables that are arrays or structures, individual array elements or structure members are written tightly packed in order. For vector types, individual components are written in order. For matrix types, outputs are written as an array of column vectors.

If any component of an output with an assigned transform feedback offset was not written to by its shader, the value recorded for that component is undefined. All components of an output variable **must** be written at an offset aligned to the size of the component. The size of each component of an output variable **must** be at least 32-bits. When capturing a vertex, any portion of the reserved storage not associated with an output variable with an assigned transform feedback offset will be unmodified.

When transform feedback is inactive, no vertices are recorded. If there is a valid counter buffer handle and counter buffer offset in the [pCounterBuffers](#) and [pCounterBufferOffsets](#) arrays, writes to the corresponding transform feedback buffer will start at the byte offset represented by the value stored in the counter buffer location.

Individual lines or triangles of a strip or fan primitive will be extracted and recorded separately. Incomplete primitives are not recorded.

When using a geometry shader that emits vertices to multiple vertex streams, a primitive will be assembled and output for each stream when there are enough vertices emitted for the output primitive type. All outputs assigned to a given transform feedback buffer are required to come from a single vertex stream.

The sizes of the transform feedback buffers are defined by the [vkCmdBindTransformFeedbackBuffersEXT pSizes](#) parameter for each of the bound buffers, or the size of the bound buffer, whichever is the lesser. If there is less space remaining in any of the transform feedback buffers than the size of the all the vertex data for that primitive based on the [XfbStride](#) for that [XfbBuffer](#) then no vertex data of that primitive is recorded in any transform feedback buffer, and the value for the number of primitives written in the corresponding [VK\\_QUERY\\_TYPE\\_TRANSFORM\\_FEEDBACK\\_STREAM\\_EXT](#) query for all transform feedback buffers is no longer incremented.

Any outputs made to a `XfbBuffer` that is not bound to a transform feedback buffer is ignored.

To bind transform feedback buffers to a command buffer for use in subsequent draw commands, call:

```
void vkCmdBindTransformFeedbackBuffersEXT(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkBuffer*  
    const VkDeviceSize*  
    const VkDeviceSize*  
                                commandBuffer,  
                                firstBinding,  
                                bindingCount,  
                                pBuffers,  
                                pOffsets,  
                                pSizes);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `firstBinding` is the index of the first transform feedback binding whose state is updated by the command.
- `bindingCount` is the number of transform feedback bindings whose state is updated by the command.
- `pBuffers` is a pointer to an array of buffer handles.
- `pOffsets` is a pointer to an array of buffer offsets.
- `pSizes` is an optional array of buffer sizes, specifying the maximum number of bytes to capture to the corresponding transform feedback buffer. If `pSizes` is `NULL`, or the value of the `pSizes` array element is `VK_WHOLE_SIZE`, then the maximum bytes captured will be the size of the corresponding buffer minus the buffer offset.

The values taken from elements `i` of `pBuffers`, `pOffsets` and `pSizes` replace the current state for the transform feedback binding `firstBinding + i`, for `i` in `[0, bindingCount]`. The transform feedback binding is updated to start at the offset indicated by `pOffsets[i]` from the start of the buffer `pBuffers[i]`.

## Valid Usage

- `VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback` **must** be enabled
- `firstBinding` **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT ::maxTransformFeedbackBuffers`
- The sum of `firstBinding` and `bindingCount` **must** be less than or equal to `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers`
- All elements of `pOffsets` **must** be less than the size of the corresponding element in `pBuffers`
- All elements of `pOffsets` **must** be a multiple of 4
- All elements of `pBuffers` **must** have been created with the `VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT` flag
- If the optional `pSize` array is specified, each element of `pSizes` **must** either be `VK_WHOLE_SIZE`, or be less than or equal to `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferSize`
- All elements of `pSizes` **must** be less than or equal to the size of the corresponding buffer in `pBuffers`
- All elements of `pOffsets` plus `pSizes`, where the `pSizes` element is not `VK_WHOLE_SIZE`, **must** be less than or equal to the size of the corresponding element in `pBuffers`
- Each element of `pBuffers` that is non-sparse **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- Transform feedback **must** not be active when the `vkCmdBindTransformFeedbackBuffersEXT` command is recorded

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pBuffers` **must** be a valid pointer to an array of `bindingCount` valid `VkBuffer` handles
- `pOffsets` **must** be a valid pointer to an array of `bindingCount` `VkDeviceSize` values
- If `pSizes` is not `NULL`, `pSizes` **must** be a valid pointer to an array of `bindingCount` `VkDeviceSize` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- If `pSizes` is not `NULL`, `bindingCount` **must** be greater than 0
- Both of `commandBuffer`, and the elements of `pBuffers` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

Transform feedback for specific transform feedback buffers is made active by calling:

```
void vkCmdBeginTransformFeedbackEXT(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkBuffer*  
    const VkDeviceSize*  
                                commandBuffer,  
                                firstCounterBuffer,  
                                counterBufferCount,  
                                pCounterBuffers,  
                                pCounterBufferOffsets);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `firstCounterBuffer` is the index of the first transform feedback buffer corresponding to `pCounterBuffers[0]` and `pCounterBufferOffsets[0]`.
- `counterBufferCount` is the size of the `pCounterBuffers` and `pCounterBufferOffsets` arrays.
- `pCounterBuffers` is an optional array of buffer handles to the counter buffers which contain a 4 byte integer value representing the byte offset from the start of the corresponding transform feedback buffer from where to start capturing vertex data. If the byte offset stored to the counter buffer location was done using `vkCmdEndTransformFeedbackEXT` it can be used to resume transform feedback from the previous location. If `pCounterBuffers` is `NULL`, then transform feedback will start capturing vertex data to byte offset zero in all bound transform feedback buffers. For each element of `pCounterBuffers` that is `VK_NULL_HANDLE`, transform feedback will start capturing vertex data to byte zero in the corresponding bound transform feedback buffer.
- `pCounterBufferOffsets` is an optional array of offsets within each of the `pCounterBuffers` where the counter values were previously written. The location in each counter buffer at these offsets **must** be large enough to contain 4 bytes of data. This data is the number of bytes captured by the previous transform feedback to this buffer. If `pCounterBufferOffsets` is `NULL`, then it is assumed the offsets are zero.

The active transform feedback buffers will capture primitives emitted from the corresponding `XfbBuffer` in the bound graphics pipeline. Any `XfbBuffer` emitted that does not output to an active

transform feedback buffer will not be captured.

## Valid Usage

- `VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback` **must** be enabled
- Transform feedback **must** not be active
- `firstCounterBuffer` **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers`
- The sum of `firstCounterBuffer` and `counterBufferCount` **must** be less than or equal to `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers`
- If `counterBufferCount` is not `0`, and `pCounterBuffers` is not `NULL`, `pCounterBuffers` **must** be a valid pointer to an array of `counterBufferCount` `VkBuffer` handles that are either valid or `VK_NULL_HANDLE`
- For each buffer handle in the array, if it is not `VK_NULL_HANDLE` it **must** reference a buffer large enough to hold 4 bytes at the corresponding offset from the `pCounterBufferOffsets` array
- If `pCounterBuffer` is `NULL`, then `pCounterBufferOffsets` **must** also be `NULL`
- For each buffer handle in the `pCounterBuffers` array that is not `VK_NULL_HANDLE` it **must** have been created with a `usage` value containing `VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT`
- Transform feedback **must** not be made active in a render pass instance with multiview enabled

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- If `counterBufferCount` is not `0`, and `pCounterBufferOffsets` is not `NULL`, `pCounterBufferOffsets` **must** be a valid pointer to an array of `counterBufferCount` `VkDeviceSize` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Both of `commandBuffer`, and the elements of `pCounterBuffers` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	

Transform feedback for specific transform feedback buffers is made inactive by calling:

```
void vkCmdEndTransformFeedbackEXT(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkBuffer*  
    const VkDeviceSize*  
                                commandBuffer,  
                                firstCounterBuffer,  
                                counterBufferCount,  
                                pCounterBuffers,  
                                pCounterBufferOffsets);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `firstCounterBuffer` is the index of the first transform feedback buffer corresponding to `pCounterBuffers[0]` and `pCounterBufferOffsets[0]`.
- `counterBufferCount` is the size of the `pCounterBuffers` and `pCounterBufferOffsets` arrays.
- `pCounterBuffers` is an optional array of buffer handles to the counter buffers used to record the current byte positions of each transform feedback buffer where the next vertex output data would be captured. This **can** be used by a subsequent `vkCmdBeginTransformFeedbackEXT` call to resume transform feedback capture from this position. It can also be used by `vkCmdDrawIndirectByteCountEXT` to determine the vertex count of the draw call.
- `pCounterBufferOffsets` is an optional array of offsets within each of the `pCounterBuffers` where the counter values can be written. The location in each counter buffer at these offsets **must** be large enough to contain 4 bytes of data. The data stored at this location is the byte offset from the start of the transform feedback buffer binding where the next vertex data would be written. If `pCounterBufferOffsets` is `NULL`, then it is assumed the offsets are zero.

## Valid Usage

- `VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback` **must** be enabled
- Transform feedback **must** be active
- `firstCounterBuffer` **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT ::maxTransformFeedbackBuffers`
- The sum of `firstCounterBuffer` and `counterBufferCount` **must** be less than or equal to `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers`
- If `counterBufferCount` is not `0`, and `pCounterBuffers` is not `NULL`, `pCounterBuffers` **must** be a valid pointer to an array of `counterBufferCount` `VkBuffer` handles that are either valid or `VK_NULL_HANDLE`
- For each buffer handle in the array, if it is not `VK_NULL_HANDLE` it **must** reference a buffer large enough to hold 4 bytes at the corresponding offset from the `pCounterBufferOffsets` array
- If `pCounterBuffer` is `NULL`, then `pCounterBufferOffsets` **must** also be `NULL`
- For each buffer handle in the `pCounterBuffers` array that is not `VK_NULL_HANDLE` it **must** have been created with a `usage` value containing `VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- If `counterBufferCount` is not `0`, and `pCounterBufferOffsets` is not `NULL`, `pCounterBufferOffsets` **must** be a valid pointer to an array of `counterBufferCount` `VkDeviceSize` values
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- Both of `commandBuffer`, and the elements of `pCounterBuffers` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics	

## 25.2. Viewport Swizzle

Each primitive sent to a given viewport has a swizzle and **optional** negation applied to its clip coordinates. The swizzle that is applied depends on the viewport index, and is controlled by the `VkPipelineViewportSwizzleStateCreateInfoNV` pipeline state:

```
typedef struct VkPipelineViewportSwizzleStateCreateInfoNV {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineViewportSwizzleStateCreateFlagsNV   flags;
    uint32_t                  viewportCount;
    const VkViewportSwizzleNV* pViewportSwizzles;
} VkPipelineViewportSwizzleStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `viewportCount` is the number of viewport swizzles used by the pipeline.
- `pViewportSwizzles` is a pointer to an array of `VkViewportSwizzleNV` structures, defining the viewport swizzles.

### Valid Usage

- `viewportCount` **must** match the `viewportCount` set in `VkPipelineViewportStateCreateInfo`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SWIZZLE_STATE_CREATE_INFO_NV`
- `flags` **must** be `0`
- `pViewportSwizzles` **must** be a valid pointer to an array of `viewportCount` valid `VkViewportSwizzleNV` structures
- `viewportCount` **must** be greater than `0`

```
typedef VkFlags VkPipelineViewportSwizzleStateCreateFlagsNV;
```

`VkPipelineViewportSwizzleStateCreateFlagsNV` is a bitmask type for setting a mask, but is currently reserved for future use.

The `VkPipelineViewportSwizzleStateCreateInfoNV` state is set by adding an instance of this structure to the `pNext` chain of an instance of the `VkPipelineViewStateCreateInfo` structure and setting the graphics pipeline state with `vkCreateGraphicsPipelines`.

Each viewport specified from 0 to `viewportCount` - 1 has its x,y,z,w swizzle state set to the corresponding `x`, `y`, `z` and `w` in the `VkViewportSwizzleNV` structure. Each component is of type `VkViewportCoordinateSwizzleNV`, which determines the type of swizzle for that component. The value of `x` computes the new x component of the position as:

```
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_X_NV) x' = x;  
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_X_NV) x' = -x;  
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Y_NV) x' = y;  
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Y_NV) x' = -y;  
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Z_NV) x' = z;  
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Z_NV) x' = -z;  
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_W_NV) x' = w;  
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_W_NV) x' = -w;
```

Similar selections are performed for the `y`, `z`, and `w` coordinates. This swizzling is applied before clipping and perspective divide. If the swizzle for an active viewport index is not specified, the swizzle for `x` is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_X_NV`, `y` is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Y_NV`, `z` is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Z_NV` and `w` is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_W_NV`.

Viewport swizzle parameters are specified by setting the `pNext` pointer of `VkGraphicsPipelineCreateInfo` to point to an instance of `VkPipelineViewportSwizzleStateCreateInfoNV`. `VkPipelineViewportSwizzleStateCreateInfoNV` uses `VkViewportSwizzleNV` to set the viewport swizzle parameters.

The `VkViewportSwizzleNV` structure is defined as:

```
typedef struct VkViewportSwizzleNV {  
    VkViewportCoordinateSwizzleNV    x;  
    VkViewportCoordinateSwizzleNV    y;  
    VkViewportCoordinateSwizzleNV    z;  
    VkViewportCoordinateSwizzleNV    w;  
} VkViewportSwizzleNV;
```

- `x` is a `VkViewportCoordinateSwizzleNV` value specifying the swizzle operation to apply to the `x` component of the primitive
- `y` is a `VkViewportCoordinateSwizzleNV` value specifying the swizzle operation to apply to the `y`

component of the primitive

- `z` is a [VkViewportCoordinateSwizzleNV](#) value specifying the swizzle operation to apply to the `z` component of the primitive
- `w` is a [VkViewportCoordinateSwizzleNV](#) value specifying the swizzle operation to apply to the `w` component of the primitive

### Valid Usage (Implicit)

- `x` **must** be a valid [VkViewportCoordinateSwizzleNV](#) value
- `y` **must** be a valid [VkViewportCoordinateSwizzleNV](#) value
- `z` **must** be a valid [VkViewportCoordinateSwizzleNV](#) value
- `w` **must** be a valid [VkViewportCoordinateSwizzleNV](#) value

Possible values of the `VkViewportSwizzleNV::x`, `y`, `z`, and `w` members, specifying swizzling of the corresponding components of primitives, are:

```
typedef enum VkViewportCoordinateSwizzleNV {
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_X_NV = 0,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_X_NV = 1,
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Y_NV = 2,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Y_NV = 3,
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Z_NV = 4,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Z_NV = 5,
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_W_NV = 6,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_W_NV = 7,
    VK_VIEWPORT_COORDINATE_SWIZZLE_MAX_ENUM_NV = 0x7FFFFFFF
} VkViewportCoordinateSwizzleNV;
```

These values are described in detail in [Viewport Swizzle](#).

## 25.3. Flat Shading

*Flat shading* a vertex output attribute means to assign all vertices of the primitive the same value for that output. The output values assigned are those of the *provoking vertex* of the primitive. Flat shading is applied to those vertex attributes that [match](#) fragment input attributes which are decorated as [Flat](#).

If neither [geometry](#) nor [tessellation shading](#) is active, the provoking vertex is determined by the [primitive topology](#) defined by [VkPipelineInputAssemblyStateCreateInfo:topology](#) used to execute the [drawing command](#).

If [geometry shading](#) is active, the provoking vertex is determined by the [primitive topology](#) defined by the [OutputPoints](#), [OutputLineStrips](#), or [OutputTriangleStrips](#) execution mode.

If [tessellation shading](#) is active but [geometry shading](#) is not, the provoking vertex **may** be any of the

vertices in each primitive.

## 25.4. Primitive Clipping

Primitives are culled against the *cull volume* and then clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by:

$$\begin{aligned} -w_c &\leq x_c \leq w_c \\ -w_c &\leq y_c \leq w_c \\ 0 &\leq z_c \leq w_c \end{aligned}$$

This view volume **can** be further restricted by as many as `VkPhysicalDeviceLimits::maxClipDistances` client-defined half-spaces.

The cull volume is the intersection of up to `VkPhysicalDeviceLimits::maxCullDistances` client-defined half-spaces (if no client-defined cull half-spaces are enabled, culling against the cull volume is skipped).

A shader **must** write a single cull distance for each enabled cull half-space to elements of the `CullDistance` array. If the cull distance for any enabled cull half-space is negative for all of the vertices of the primitive under consideration, the primitive is discarded. Otherwise the primitive is clipped against the clip volume as defined below.

The clip volume is the intersection of up to `VkPhysicalDeviceLimits::maxClipDistances` client-defined half-spaces with the view volume (if no client-defined clip half-spaces are enabled, the clip volume is the view volume).

A shader **must** write a single clip distance for each enabled clip half-space to elements of the `ClipDistance` array. Clip half-space  $i$  is then given by the set of points satisfying the inequality

$$c_i(\mathbf{P}) \geq 0$$

where  $c_i(\mathbf{P})$  is the clip distance  $i$  at point  $\mathbf{P}$ . For point primitives,  $c_i(\mathbf{P})$  is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections [Basic Line Segment Rasterization](#) and [Basic Polygon Rasterization](#), using the perspective interpolation equations.

The number of client-defined clip and cull half-spaces that are enabled is determined by the explicit size of the built-in arrays `ClipDistance` and `CullDistance`, respectively, declared as an output in the interface of the entry point of the final shader stage before clipping.

If `VkPipelineRasterizationDepthClipStateCreateInfoEXT` is present in the graphics pipeline state then depth clipping is disabled if `VkPipelineRasterizationDepthClipStateCreateInfoEXT::depthClipEnable` is `VK_FALSE`. Otherwise, if `VkPipelineRasterizationDepthClipStateCreateInfoEXT` is not present, depth clipping is disabled when `VkPipelineRasterizationStateCreateInfo::depthClampEnable` is `VK_TRUE`. When depth clipping is disabled, the plane equation

$$0 \leq z_c \leq w_c$$

(see the clip volume definition above) is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point or line segment, then clipping passes it unchanged if its vertices lie entirely within the clip volume.

Possible values of `VkPhysicalDevicePointClippingProperties::pointClippingBehavior`, specifying clipping behavior of a point primitive whose vertex lies outside the clip volume, are:

```
typedef enum VkPointClippingBehavior {
    VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES = 0,
    VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY = 1,
    VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES_KHR =
VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES,
    VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY_KHR =
VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY,
    VK_POINT_CLIPPING_BEHAVIOR_MAX_ENUM = 0x7FFFFFFF
} VkPointClippingBehavior;
```

or the equivalent

```
typedef VkPointClippingBehavior VkPointClippingBehaviorKHR;
```

- `VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES` specifies that the primitive is discarded if the vertex lies outside any clip plane, including the planes bounding the view volume.
- `VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY` specifies that the primitive is discarded only if the vertex lies outside any user clip plane.

If either of a line segment's vertices lie outside of the clip volume, the line segment **may** be clipped, with new vertex coordinates computed for each vertex that lies outside the clip volume. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value,  $0 \leq t \leq 1$ , for each clipped vertex. If the coordinates of a clipped vertex are  $\mathbf{P}$  and the original vertices' coordinates are  $\mathbf{P}_1$  and  $\mathbf{P}_2$ , then  $t$  is given by

$$\mathbf{P} = t \mathbf{P}_1 + (1-t) \mathbf{P}_2.$$

$t$  is used to clip vertex output attributes as described in [Clipping Shader Outputs](#).

If the primitive is a polygon, it passes unchanged if every one of its edges lie entirely inside the clip volume, and it is discarded if every one of its edges lie entirely outside the clip volume. If the edges of the polygon intersect the boundary of the clip volume, the intersecting edges are reconnected by new edges that lie along the boundary of the clip volume - in some cases requiring the introduction of new vertices into a polygon.

If a polygon intersects an edge of the clip volume's boundary, the clipped polygon **must** include a point on this boundary edge.

Primitives rendered with user-defined half-spaces **must** satisfy a complementarity criterion. Suppose a series of primitives is drawn where each vertex  $i$  has a single specified clip distance  $d_i$  (or a number of similarly specified clip distances, if multiple half-spaces are enabled). Next, suppose that the same series of primitives are drawn again with each such clip distance replaced by  $-d_i$  (and the graphics pipeline is otherwise the same). In this case, primitives **must** not be missing any pixels, and pixels **must** not be drawn twice in regions where those primitives are cut by the clip planes.

## 25.5. Clipping Shader Outputs

Next, vertex output attributes are clipped. The output values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the output values assigned to vertices produced by clipping are clipped.

Let the output values assigned to the two vertices  $P_1$  and  $P_2$  of an unclipped edge be  $c_1$  and  $c_2$ . The value of  $t$  (see [Primitive Clipping](#)) for a clipped point  $P$  is used to obtain the output value associated with  $P$  as

$$c = t c_1 + (1-t) c_2.$$

(Multiplying an output value by a scalar means multiplying each of  $x, y, z$ , and  $w$  by the scalar.)

Since this computation is performed in clip space before division by  $w_c$ , clipped output values are perspective-correct.

Polygon clipping creates a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one half-space at a time. Output value clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

For vertex output attributes whose matching fragment input attributes are decorated with [NoPerspective](#), the value of  $t$  used to obtain the output value associated with  $P$  will be adjusted to produce results that vary linearly in framebuffer space.

Output attributes of integer or unsigned integer type **must** always be flat shaded. Flat shaded attributes are constant over the primitive being rasterized (see [Basic Line Segment Rasterization](#) and [Basic Polygon Rasterization](#)), and no interpolation is performed. The output value  $c$  is taken from either  $c_1$  or  $c_2$ , since flat shading has already occurred and the two values are identical.

## 25.6. Controlling Viewport W Scaling

If viewport **W** scaling is enabled, the **W** component of the clip coordinate is modified by the provided coefficients from the corresponding viewport as follows.

$$w'_c = x_{\text{coeff}} x_c + y_{\text{coeff}} y_c + w_c$$

The [`VkPipelineViewportWScaleCreateInfoNV`](#) structure is defined as:

```

typedef struct VkPipelineViewportWSizingStateCreateInfoNV {
    VkStructureType          sType;
    const void*             pNext;
    VkBool32                 viewportWSizingEnable;
    uint32_t                 viewportCount;
    const VkViewportWSizingNV* pViewportWScalings;
} VkPipelineViewportWSizingStateCreateInfoNV;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **viewportWSizingEnable** controls whether viewport **W** scaling is enabled.
- **viewportCount** is the number of viewports used by **W** scaling, and **must** match the number of viewports in the pipeline if viewport **W** scaling is enabled.
- **pViewportWScalings** is a pointer to an array of **VkViewportWSizingNV** structures defining the **W** scaling parameters for the corresponding viewports. If the viewport **W** scaling state is dynamic, this member is ignored.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_PIPELINE\_VIEWPORT\_W\_SCALING\_STATE\_CREATE\_INFO\_NV**
- **viewportCount** **must** be greater than **0**

The **VkPipelineViewportWSizingStateCreateInfoNV** state is set by adding an instance of this structure to the **pNext** chain of an instance of the **VkPipelineViewStateCreateInfo** structure and setting the graphics pipeline state with **vkCreateGraphicsPipelines**.

If the bound pipeline state object was not created with the **VK\_DYNAMIC\_STATE\_VIEWPORT\_W\_SCALING\_NV** dynamic state enabled, viewport **W** scaling parameters are specified using the **pViewportWScalings** member of **VkPipelineViewportWSizingStateCreateInfoNV** in the pipeline state object. If the pipeline state object was created with the **VK\_DYNAMIC\_STATE\_VIEWPORT\_W\_SCALING\_NV** dynamic state enabled, the viewport transformation parameters are dynamically set and changed with the command:

```

void vkCmdSetViewportWSizingNV(
    VkCommandBuffer                      commandBuffer,
    uint32_t                            firstViewport,
    uint32_t                            viewportCount,
    const VkViewportWSizingNV*         pViewportWScalings);

```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **firstViewport** is the index of the first viewport whose parameters are updated by the command.
- **viewportCount** is the number of viewports whose parameters are updated by the command.
- **pViewportWScalings** is a pointer to an array of **VkViewportWSizingNV** structures specifying

viewport parameters.

The viewport parameters taken from element  $i$  of `pViewportWScalings` replace the current state for the viewport index `firstViewport + i`, for  $i$  in  $[0, \text{viewportCount}]$ .

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic state enabled
- `firstViewport` **must** be less than `VkPhysicalDeviceLimits::maxViewports`
- The sum of `firstViewport` and `viewportCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pViewportWScalings` **must** be a valid pointer to an array of `viewportCount` `VkViewportScalingNV` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `viewportCount` **must** be greater than 0

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

Both `VkPipelineViewportScalingStateCreateInfoNV` and `vkCmdSetViewportScalingNV` use `VkViewportScalingNV` to set the viewport transformation parameters.

The `VkViewportScalingNV` structure is defined as:

```

typedef struct VkViewportWScalingNV {
    float     xcoeff;
    float     ycoeff;
} VkViewportWScalingNV;

```

- **xcoeff** and **ycoeff** are the viewport's W scaling factor for x and y respectively.

## 25.7. Coordinate Transformations

*Clip coordinates* for a vertex result from shader execution, which yields a vertex coordinate **Position**.

Perspective division on clip coordinates yields *normalized device coordinates*, followed by a *viewport* transformation (see [Controlling the Viewport](#)) to convert these coordinates into *framebuffer coordinates*.

If a vertex in clip coordinates has a position given by

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

then the vertex's normalized device coordinates are

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

## 25.8. Controlling the Viewport

The viewport transformation is determined by the selected viewport's width and height in pixels,  $p_x$  and  $p_y$ , respectively, and its center ( $o_x$ ,  $o_y$ ) (also in pixels), as well as its depth range min and max determining a depth range scale value  $p_z$  and a depth range bias value  $o_z$  (defined below). The vertex's framebuffer coordinates ( $x_f$ ,  $y_f$ ,  $z_f$ ) are given by

$$x_f = (p_x / 2) x_d + o_x$$

$$y_f = (p_y / 2) y_d + o_y$$

$$z_f = p_z \times z_d + o_z$$

Multiple viewports are available, numbered zero up to [VkPhysicalDeviceLimits::maxViewports](#) minus one. The number of viewports used by a pipeline is controlled by the [viewportCount](#) member of the [VkPipelineViewportStateCreateInfo](#) structure used in pipeline creation.

The [VkPipelineViewportStateCreateInfo](#) structure is defined as:

```

typedef struct VkPipelineViewportStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineViewportStateCreateFlags flags;
    uint32_t viewportCount;
    const VkViewport* pViewports;
    uint32_t scissorCount;
    const VkRect2D* pScissors;
} VkPipelineViewportStateCreateInfo;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **viewportCount** is the number of viewports used by the pipeline.
- **pViewports** is a pointer to an array of **VkViewport** structures, defining the viewport transforms. If the viewport state is dynamic, this member is ignored.
- **scissorCount** is the number of **scissors** and **must** match the number of viewports.
- **pScissors** is a pointer to an array of **VkRect2D** structures defining the rectangular bounds of the scissor for the corresponding viewport. If the scissor state is dynamic, this member is ignored.

## Valid Usage

- If the **multiple viewports** feature is not enabled, **viewportCount must be 1**
- If the **multiple viewports** feature is not enabled, **scissorCount must be 1**
- **viewportCount must be between 1 and **VkPhysicalDeviceLimits::maxViewports**, inclusive**
- **scissorCount must be between 1 and **VkPhysicalDeviceLimits::maxViewports**, inclusive**
- **scissorCount and viewportCount must be identical**
- The **x** and **y** members of **offset** member of any element of **pScissors must be greater than or equal to 0**
- Evaluation of **(offset.x + extent.width) must not cause a signed integer addition overflow for any element of pScissors**
- Evaluation of **(offset.y + extent.height) must not cause a signed integer addition overflow for any element of pScissors**
- If the **viewportWScaleEnable** member of a **VkPipelineViewportWScaleStateCreateInfoNV** structure chained to the **pNext** chain is **VK\_TRUE**, the **viewportCount** member of the **VkPipelineViewportWScaleStateCreateInfoNV** structure **must be equal to viewportCount**

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkPipelineViewportCoarseSampleOrderStateCreateInfoNV`, `VkPipelineViewportExclusiveScissorStateCreateInfoNV`, `VkPipelineViewportShadingRateImageStateCreateInfoNV`, `VkPipelineViewportSwizzleStateCreateInfoNV`, `VkPipelineViewportWScalingStateCreateInfoNV` or `VkPipelineViewportWSampleOrderStateCreateInfoNV`
- Each `sType` member in the `pNext` chain must be unique
- `flags` must be `0`
- `viewportCount` must be greater than `0`
- `scissorCount` must be greater than `0`

```
typedef VkFlags VkPipelineViewportStateCreateInfo;
```

`VkPipelineViewportStateCreateInfo` is a bitmask type for setting a mask, but is currently reserved for future use.

A *vertex processing stage* can direct each primitive to zero or more viewports. The destination viewports for a primitive are selected by the last active vertex processing stage that has an output variable decorated with `ViewportIndex` (selecting a single viewport) or `ViewportMaskNV` (selecting multiple viewports). The viewport transform uses the viewport corresponding to either the value assigned to `ViewportIndex` or one of the bits set in `ViewportMaskNV`, and taken from an implementation-dependent vertex of each primitive. If `ViewportIndex` or any of the bits in `ViewportMaskNV` are outside the range zero to `viewportCount` minus one for a primitive, or if the last active vertex processing stage did not assign a value to either `ViewportIndex` or `ViewportMaskNV` for all vertices of a primitive due to flow control, the values resulting from the viewport transformation of the vertices of such primitives are undefined. If the last vertex processing stage does not have an output decorated with `ViewportIndex` or `ViewportMaskNV`, the viewport numbered zero is used by the viewport transformation.

A single vertex can be used in more than one individual primitive, in primitives such as `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`. In this case, the viewport transformation is applied separately for each primitive.

If the bound pipeline state object was not created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled, viewport transformation parameters are specified using the `pViewports` member of `VkPipelineViewportStateCreateInfo` in the pipeline state object. If the pipeline state object was created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled, the viewport transformation parameters are dynamically set and changed with the command:

```
void vkCmdSetViewport(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkViewport*  
                                commandBuffer,  
                                firstViewport,  
                                viewportCount,  
                                pViewports);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstViewport` is the index of the first viewport whose parameters are updated by the command.
- `viewportCount` is the number of viewports whose parameters are updated by the command.
- `pViewports` is a pointer to an array of `VkViewport` structures specifying viewport parameters.

The viewport parameters taken from element `i` of `pViewports` replace the current state for the viewport index `firstViewport + i`, for `i` in `[0, viewportCount]`.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled
- `firstViewport` **must** be less than `VkPhysicalDeviceLimits::maxViewports`
- The sum of `firstViewport` and `viewportCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- If the `multiple viewports` feature is not enabled, `firstViewport` **must** be 0
- If the `multiple viewports` feature is not enabled, `viewportCount` **must** be 1

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pViewports` **must** be a valid pointer to an array of `viewportCount` valid `VkViewport` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `viewportCount` **must** be greater than 0

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

Both `VkPipelineViewportStateCreateInfo` and `vkCmdSetViewport` use `VkViewport` to set the viewport transformation parameters.

The `VkViewport` structure is defined as:

```
typedef struct VkViewport {  
    float    x;  
    float    y;  
    float    width;  
    float    height;  
    float    minDepth;  
    float    maxDepth;  
} VkViewport;
```

- `x` and `y` are the viewport's upper left corner (x,y).
- `width` and `height` are the viewport's width and height, respectively.
- `minDepth` and `maxDepth` are the depth range for the viewport. It is valid for `minDepth` to be greater than or equal to `maxDepth`.

The framebuffer depth coordinate  $z_f$  **may** be represented using either a fixed-point or floating-point representation. However, a floating-point representation **must** be used if the depth/stencil attachment has a floating-point depth component. If an m-bit fixed-point representation is used, we assume that it represents each value  $\frac{k}{2^m - 1}$ , where  $k \in \{0, 1, \dots, 2^m - 1\}$ , as k (e.g. 1.0 is represented in binary as a string of all ones).

The viewport parameters shown in the above equations are found from these values as

$$o_x = x + \text{width} / 2$$

$$o_y = y + \text{height} / 2$$

$$o_z = \text{minDepth}$$

$$p_x = \text{width}$$

$$p_y = \text{height}$$

$$p_z = \text{maxDepth} - \text{minDepth}.$$

The application **can** specify a negative term for `height`, which has the effect of negating the `y` coordinate in clip space before performing the transform. When using a negative `height`, the application **should** also adjust the `y` value to point to the lower left corner of the viewport instead of the upper left corner. Using the negative `height` allows the application to avoid having to negate the `y` component of the `Position` output from the last vertex processing stage in shaders that also target other graphics APIs.

The width and height of the `implementation-dependent maximum viewport dimensions` **must** be greater than or equal to the width and height of the largest image which **can** be created and attached to a framebuffer.

The floating-point viewport bounds are represented with an `implementation-dependent precision`.

## Valid Usage

- `width` **must** be greater than `0.0`
- `width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions[0]`
- The absolute value of `height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions[1]`
- `x` **must** be greater than or equal to `viewportBoundsRange[0]`
- `(x + width)` **must** be less than or equal to `viewportBoundsRange[1]`
- `y` **must** be greater than or equal to `viewportBoundsRange[0]`
- `y` **must** be less than or equal to `viewportBoundsRange[1]`
- `(y + height)` **must** be greater than or equal to `viewportBoundsRange[0]`
- `(y + height)` **must** be less than or equal to `viewportBoundsRange[1]`
- Unless `VK_EXT_depth_range_unrestricted` extension is enabled `minDepth` **must** be between `0.0` and `1.0`, inclusive
- Unless `VK_EXT_depth_range_unrestricted` extension is enabled `maxDepth` **must** be between `0.0` and `1.0`, inclusive

# Chapter 26. Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains associated data such as depth, color, or other attributes.

Rasterizing a primitive begins by determining which squares of an integer grid in framebuffer coordinates are occupied by the primitive, and assigning one or more depth values to each such square. This process is described below for points, lines, and polygons.

A grid square, including its (x,y) framebuffer coordinates, z (depth), and associated data added by fragment shaders, is called a fragment. A fragment is located by its upper left corner, which lies on integer grid coordinates.

Rasterization operations also refer to a fragment's sample locations, which are offset by fractional values from its upper left corner. The rasterization rules for points, lines, and triangles involve testing whether each sample location is inside the primitive. Fragments need not actually be square, and rasterization rules are not affected by the aspect ratio of fragments. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other.

We assume that fragments are square, since it simplifies antialiasing and texturing. After rasterization, fragments are processed by the [early per-fragment tests](#), if enabled.

Several factors affect rasterization, including the members of [VkPipelineRasterizationStateCreateInfo](#) and [VkPipelineMultisampleStateCreateInfo](#).

The [VkPipelineRasterizationStateCreateInfo](#) structure is defined as:

```
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkPipelineRasterizationStateCreateInfoFlags   flags;
    VkBool32                  depthClampEnable;
    VkBool32                  rasterizerDiscardEnable;
    VkPolygonMode              polygonMode;
    VkCullModeFlags            cullMode;
    VkFrontFace                frontFace;
    VkBool32                  depthBiasEnable;
    float                     depthBiasConstantFactor;
    float                     depthBiasClamp;
    float                     depthBiasSlopeFactor;
    float                     lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.

- `depthClampEnable` controls whether to clamp the fragment's depth values as described in [Depth Test](#). If the pipeline is not created with `VkPipelineRasterizationDepthClipStateCreateInfoEXT` present then enabling depth clamp will also disable clipping primitives to the z planes of the frustum as described in [Primitive Clipping](#). Otherwise depth clipping is controlled by the state set in `VkPipelineRasterizationDepthClipStateCreateInfoEXT`.
- `rasterizerDiscardEnable` controls whether primitives are discarded immediately before the rasterization stage.
- `polygonMode` is the triangle rendering mode. See [VkPolygonMode](#).
- `cullMode` is the triangle facing direction used for primitive culling. See [VkCullModeFlagBits](#).
- `frontFace` is a [VkFrontFace](#) value specifying the front-facing triangle orientation to be used for culling.
- `depthBiasEnable` controls whether to bias fragment depth values.
- `depthBiasConstantFactor` is a scalar factor controlling the constant depth value added to each fragment.
- `depthBiasClamp` is the maximum (or minimum) depth bias of a fragment.
- `depthBiasSlopeFactor` is a scalar factor applied to a fragment's slope in depth bias calculations.
- `lineWidth` is the width of rasterized line segments.

The application **can** also add a `VkPipelineRasterizationStateRasterizationOrderAMD` structure to the `pNext` chain of a `VkPipelineRasterizationStateCreateInfo` structure. This structure enables selecting the rasterization order to use when rendering with the corresponding graphics pipeline as described in [Rasterization Order](#).

## Valid Usage

- If the `depth clamping` feature is not enabled, `depthClampEnable` **must** be `VK_FALSE`
- If the `non-solid fill modes` feature is not enabled, `polygonMode` **must** be `VK_POLYGON_MODE_FILL` or `VK_POLYGON_MODE_FILL_RECTANGLE_NV`
- If the `VK_NV_fill_rectangle` extension is not enabled, `polygonMode` **must** not be `VK_POLYGON_MODE_FILL_RECTANGLE_NV`

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO`
- Each **pNext** member of any structure (including this one) in the **pNext** chain must be either `NULL` or a pointer to a valid instance of `VkPipelineRasterizationConservativeStateCreateInfoEXT`, `VkPipelineRasterizationDepthClipStateCreateInfoEXT`, `VkPipelineRasterizationLineStateCreateInfoEXT`, `VkPipelineRasterizationStateRasterizationOrderAMD`, `VkPipelineRasterizationStateStreamCreateInfoEXT` or
- Each **sType** member in the **pNext** chain must be unique
- **flags** must be `0`
- **polygonMode** must be a valid `VkPolygonMode` value
- **cullMode** must be a valid combination of `VkCullModeFlagBits` values
- **frontFace** must be a valid `VkFrontFace` value

```
typedef VkFlags VkPipelineRasterizationStateCreateInfo;
```

`VkPipelineRasterizationStateCreateInfo` is a bitmask type for setting a mask, but is currently reserved for future use.

If the **pNext** chain of `VkPipelineRasterizationStateCreateInfo` includes a `VkPipelineRasterizationDepthClipStateCreateInfoEXT` structure, then that structure controls whether depth clipping is enabled or disabled.

The `VkPipelineRasterizationDepthClipStateCreateInfoEXT` structure is defined as:

```
typedef struct VkPipelineRasterizationDepthClipStateCreateInfoEXT {  
    VkStructureType sType;  
    const void* pNext;  
    VkPipelineRasterizationDepthClipStateCreateFlagsEXT flags;  
    VkBool32 depthClipEnable;  
} VkPipelineRasterizationDepthClipStateCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **depthClipEnable** controls whether depth clipping is enabled as described in [Primitive Clipping](#).

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT`
- `flags` must be 0

```
typedef VkFlags VkPipelineRasterizationDepthClipStateCreateInfoEXT;
```

`VkPipelineRasterizationDepthClipStateCreateInfoEXT` is a bitmask type for setting a mask, but is currently reserved for future use.

The `VkPipelineMultisampleStateCreateInfo` structure is defined as:

```
typedef struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkPipelineMultisampleStateCreateFlags   flags;
    VkSampleCountFlagBits      rasterizationSamples;
    VkBool32                  sampleShadingEnable;
    float                     minSampleShading;
    const VkSampleMask*        pSampleMask;
    VkBool32                  alphaToCoverageEnable;
    VkBool32                  alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `rasterizationSamples` is a `VkSampleCountFlagBits` specifying the number of samples used in rasterization.
- `sampleShadingEnable` can be used to enable [Sample Shading](#).
- `minSampleShading` specifies a minimum fraction of sample shading if `sampleShadingEnable` is set to `VK_TRUE`.
- `pSampleMask` is a bitmask of static coverage information that is ANDed with the coverage information generated during rasterization, as described in [Sample Mask](#).
- `alphaToCoverageEnable` controls whether a temporary coverage value is generated based on the alpha component of the fragment's first color output as specified in the [Multisample Coverage](#) section.
- `alphaToOneEnable` controls whether the alpha component of the fragment's first color output is replaced with one as described in [Multisample Coverage](#).

## Valid Usage

- If the `sample rate shading` feature is not enabled, `sampleShadingEnable` **must** be `VK_FALSE`
- If the `alpha to one` feature is not enabled, `alphaToOneEnable` **must** be `VK_FALSE`
- `minSampleShading` **must** be in the range [0,1]
- If the `VK_NV_framebuffer_mixed_samples` extension is enabled, and if the subpass has any color attachments and `rasterizationSamples` is greater than the number of color samples, then `sampleShadingEnable` **must** be `VK_FALSE`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkPipelineCoverageModulationStateCreateInfoNV`, `VkPipelineCoverageReductionStateCreateInfoNV`, `VkPipelineCoverageToColorStateCreateInfoNV`, `VkPipelineSampleLocationsStateCreateInfoEXT` or
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be `0`
- `rasterizationSamples` **must** be a valid `VkSampleCountFlagBits` value
- If `pSampleMask` is not `NULL`, `pSampleMask` **must** be a valid pointer to an array of  $\lceil \frac{\text{rasterizationSamples}}{32} \rceil$  `VkSampleMask` values

```
typedef VkFlags VkPipelineMultisampleStateCreateFlags;
```

`VkPipelineMultisampleStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

Rasterization only generates fragments which cover one or more pixels inside the framebuffer. Pixels outside the framebuffer are never considered covered in the fragment. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the pipeline, including any of the early per-fragment tests described in [Early Per-Fragment Tests](#).

Surviving fragments are processed by fragment shaders. Fragment shaders determine associated data for fragments, and **can** also modify or replace their assigned depth values.

When the `VK_AMD_mixed_attachment_samples` and `VK_NV_framebuffer_mixed_samples` extensions are not enabled, if the subpass for which this pipeline is being created uses color and/or depth/stencil attachments, then `rasterizationSamples` **must** be the same as the sample count for those subpass attachments.

When the `VK_AMD_mixed_attachment_samples` extension is enabled, if the subpass for which this pipeline is being created uses color and/or depth/stencil attachments, then `rasterizationSamples` **must** be the same as the maximum of the sample counts of those subpass attachments.

When the `VK_NV_framebuffer_mixed_samples` extension is enabled, `rasterizationSamples` **must** match the sample count of the depth/stencil attachment if present, otherwise **must** be greater than or equal to the sample count of the color attachments, if present.

If the `VK_NV_coverage_reduction_mode` extension is enabled, the coverage reduction mode specified by `VkPipelineCoverageReductionStateCreateInfoNV::coverageReductionMode`, the `rasterizationSamples` member of `pMultisampleState` and the sample counts for the color and depth/stencil attachments (if present) **must** be a valid combination returned by `vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV`

If the subpass for which this pipeline is being created does not use color or depth/stencil attachments, `rasterizationSamples` **must** follow the rules for a `zero-attachment` subpass.

## 26.1. Discarding Primitives Before Rasterization

Primitives are discarded before rasterization if the `rasterizerDiscardEnable` member of `VkPipelineRasterizationStateCreateInfo` is enabled. When enabled, primitives are discarded after they are processed by the last active shader stage in the pipeline before rasterization.

## 26.2. Controlling the Vertex Stream Used for Rasterization

By default vertex data output from the last vertex processing stage are directed to vertex stream zero. Geometry shaders **can** emit primitives to multiple independent vertex streams. Each vertex emitted by the geometry shader is directed at one of the vertex streams. As vertices are received on each vertex stream, they are arranged into primitives of the type specified by the geometry shader output primitive type. The shading language instructions `OpEndPrimitive` and `OpEndStreamPrimitive` **can** be used to end the primitive being assembled on a given vertex stream and start a new empty primitive of the same type. An implementation supports up to `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams` streams, which is at least 1. The individual streams are numbered 0 through `maxTransformFeedbackStreams` minus 1. There is no requirement on the order of the streams to which vertices are emitted, and the number of vertices emitted to each vertex stream **can** be completely independent, subject only to the `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreamDataSize` and `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferSizeDataSize` limits. The primitives output from all vertex streams are passed to the transform feedback stage to be captured to transform feedback buffers in the manner specified by the last vertex processing stage shader's `XfbBuffer`, `XfbStride`, and `Offsets` decorations on the output interface variables in the graphics pipeline. To use a vertex stream other than zero, or to use multiple streams, the `GeometryStreams` capability **must** be specified.

By default, the primitives output from vertex stream zero are rasterized. If the implementation supports the `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackRasterizationStreamSelect` property it is possible to rasterize a vertex stream

other than zero.

By default, geometry shaders that emit vertices to multiple vertex streams are limited to using only the `OutputPoints` output primitive type. If the implementation supports the `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackStreamsLinesTriangles` property it is possible to emit `OutputLineStrip` or `OutputTriangleStrip` in addition to `OutputPoints`.

The vertex stream used for rasterization is specified by adding a `VkPipelineRasterizationStateStreamCreateInfoEXT` structure to the `pNext` chain of a `VkPipelineRasterizationStateCreateInfo` structure.

The `VkPipelineRasterizationStateStreamCreateInfoEXT` structure is defined as:

```
typedef struct VkPipelineRasterizationStateStreamCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationStateStreamCreateFlagsEXT flags;
    uint32_t rasterizationStream;
} VkPipelineRasterizationStateStreamCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `rasterizationStream` is the vertex stream selected for rasterization.

If this structure is not present, `rasterizationStream` is assumed to be zero.

## Valid Usage

- `VkPhysicalDeviceTransformFeedbackFeaturesEXT::geometryStreams` **must** be enabled
- `rasterizationStream` **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams`
- `rasterizationStream` **must** be zero if `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackRasterizationStreamSelect` is `VK_FALSE`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_STREAM_CREATE_INFO_EXT`
- `flags` **must** be `0`

```
typedef VkFlags VkPipelineRasterizationStateStreamCreateFlagsEXT;
```

`VkPipelineRasterizationStateStreamCreateFlagsEXT` is a bitmask type for setting a mask, but is

currently reserved for future use.

## 26.3. Rasterization Order

Within a subpass of a [render pass instance](#), for a given (x,y,layer,sample) sample location, the following operations are guaranteed to execute in *rasterization order*, for each separate primitive that includes that sample location:

1. [Scissor test](#)
2. [Exclusive scissor test](#)
3. [Sample mask generation](#)
4. [Depth bounds test](#)
5. [Stencil test, stencil op and stencil write](#)
6. [Depth test and depth write](#)
7. [Sample counting for occlusion queries](#)
8. [Fragment Coverage To Color](#)
9. [coverage reduction](#)
10. [Blending, logic operations](#), and color writes

Each of these operations is atomically executed for each primitive and sample location.

Execution of these operations for each primitive in a subpass occurs in an order determined by the application.

The rasterization order to use for a graphics pipeline is specified by adding a [VkPipelineRasterizationStateRasterizationOrderAMD](#) structure to the [pNext](#) chain of a [VkPipelineRasterizationStateCreateInfo](#) structure.

The [VkPipelineRasterizationStateRasterizationOrderAMD](#) structure is defined as:

```
typedef struct VkPipelineRasterizationStateRasterizationOrderAMD {  
    VkStructureType          sType;  
    const void*              pNext;  
    VkRasterizationOrderAMD rasterizationOrder;  
} VkPipelineRasterizationStateRasterizationOrderAMD;
```

- [sType](#) is the type of this structure.
- [pNext](#) is [NULL](#) or a pointer to an extension-specific structure.
- [rasterizationOrder](#) is a [VkRasterizationOrderAMD](#) value specifying the primitive rasterization order to use.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_RASTERIZATION_ORDER_AMD`
- `rasterizationOrder` must be a valid `VkRasterizationOrderAMD` value

If the `VK_AMD_rasterization_order` device extension is not enabled or the application does not request a particular rasterization order through specifying a `VkPipelineRasterizationStateRasterizationOrderAMD` structure then the rasterization order used by the graphics pipeline defaults to `VK_RASTERIZATION_ORDER_STRICT_AMD`.

Possible values of `VkPipelineRasterizationStateRasterizationOrderAMD::rasterizationOrder`, specifying the primitive rasterization order, are:

```
typedef enum VkRasterizationOrderAMD {
    VK_RASTERIZATION_ORDER_STRICT_AMD = 0,
    VK_RASTERIZATION_ORDER_RELAXED_AMD = 1,
    VK_RASTERIZATION_ORDER_MAX_ENUM_AMD = 0x7FFFFFFF
} VkRasterizationOrderAMD;
```

- `VK_RASTERIZATION_ORDER_STRICT_AMD` specifies that operations for each primitive in a subpass **must** occur in `primitive order`.
- `VK_RASTERIZATION_ORDER_RELAXED_AMD` specifies that operations for each primitive in a subpass **may** not occur in `primitive order`.

## 26.4. Multisampling

Multisampling is a mechanism to antialias all Vulkan primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. Each sample in each framebuffer attachment has storage for a color, depth, and/or stencil value, such that per-fragment operations apply to each sample independently. The color sample values **can** be later *resolved* to a single color (see [Resolving Multisample Images](#) and the [Render Pass](#) chapter for more details on how to resolve multisample images to non-multisample images).

Vulkan defines rasterization rules for single-sample modes in a way that is equivalent to a multisample mode with a single sample in the center of each fragment.

Each fragment includes a coverage value with `rasterizationSamples` bits (see [Sample Mask](#)). Each fragment includes `rasterizationSamples` depth values and sets of associated data. An implementation **may** choose to assign the same associated data to more than one sample. The location for evaluating such associated data **may** be anywhere within the fragment area including the fragment's center location ( $x_f, y_f$ ) or any of the sample locations. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment's center location **must** be used. The different associated data values need not all be evaluated at the same location. Each fragment thus consists of integer  $x$  and  $y$  grid coordinates, `rasterizationSamples` depth values and sets of associated data, and a coverage value with `rasterizationSamples` bits.

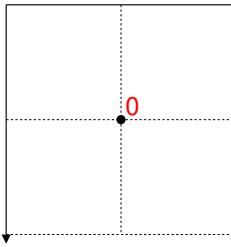
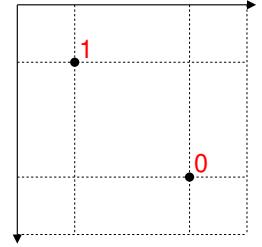
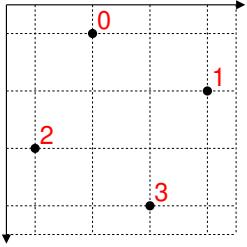
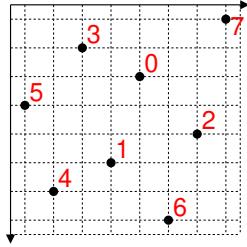
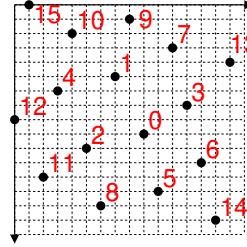
It is understood that each pixel has `rasterizationSamples` locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel **must** be located inside or on the boundary of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points **may** be identical for each pixel in the framebuffer, or they **may** differ.

If the render pass has a fragment density map attachment, each fragment only has `rasterizationSamples` locations associated with it regardless of how many pixels are covered in the fragment area. Fragment sample locations are defined as if the fragment had an area of (1,1) and its sample points **must** be located within these bounds. Their actual location in the framebuffer is calculated by scaling the sample location by the fragment area. Attachments with storage for multiple samples per pixel are located at the pixel sample locations. Otherwise, the fragment's sample locations are generally used for evaluation of associated data and fragment operations.

If the current pipeline includes a fragment shader with one or more variables in its interface decorated with `Sample` and `Input`, the data associated with those variables will be assigned independently for each sample. The values for each sample **must** be evaluated at the location of the sample. The data associated with any other variables not decorated with `Sample` and `Input` need not be evaluated independently for each sample.

If the `standardSampleLocations` member of `VkPhysicalDeviceLimits` is `VK_TRUE`, then the sample counts `VK_SAMPLE_COUNT_1_BIT`, `VK_SAMPLE_COUNT_2_BIT`, `VK_SAMPLE_COUNT_4_BIT`, `VK_SAMPLE_COUNT_8_BIT`, and `VK_SAMPLE_COUNT_16_BIT` have sample locations as listed in the following table, with the  $i$ th entry in the table corresponding to bit  $i$  in the sample masks. `VK_SAMPLE_COUNT_32_BIT` and `VK_SAMPLE_COUNT_64_BIT` do not have standard sample locations. Locations are defined relative to an origin in the upper left corner of the fragment.

Table 32. Standard sample locations

VK_SAMPLE_COUNT_1_BIT	VK_SAMPLE_COUNT_2_BIT	VK_SAMPLE_COUNT_4_BIT	VK_SAMPLE_COUNT_8_BIT	VK_SAMPLE_COUNT_16_BIT
(0.5,0.5)	(0.75,0.75) (0.25,0.25)	(0.375, 0.125) (0.875, 0.375) (0.125, 0.625) (0.625, 0.875)	(0.5625, 0.3125) (0.4375, 0.6875) (0.8125, 0.5625) (0.3125, 0.1875) (0.1875, 0.8125) (0.0625, 0.4375) (0.6875, 0.9375) (0.9375, 0.0625)	(0.5625, 0.5625) (0.4375, 0.3125) (0.3125, 0.625) (0.75, 0.4375) (0.1875, 0.375) (0.625, 0.8125) (0.8125, 0.6875) (0.6875, 0.1875) (0.375, 0.875) (0.5, 0.0625) (0.25, 0.125) (0.125, 0.75) (0.0, 0.5) (0.9375, 0.25) (0.875, 0.9375) (0.0625, 0.0)
				
				

Color images created with multiple samples per pixel use a compression technique where there are two arrays of data associated with each pixel. The first array contains one element per sample where each element stores an index to the second array defining the *fragment mask* of the pixel. The second array contains one element per *color fragment* and each element stores a unique color value in the format of the image. With this compression technique it is not always necessary to actually use unique storage locations for each color sample: when multiple samples share the same color value the fragment mask **may** have two samples referring to the same color fragment. The number of color fragments is determined by the `samples` member of the `VkImageCreateInfo` structure used to create the image. The `VK_AMD_shader_fragment_mask` device extension provides shader instructions enabling the application to get direct access to the fragment mask and the individual color fragment values.

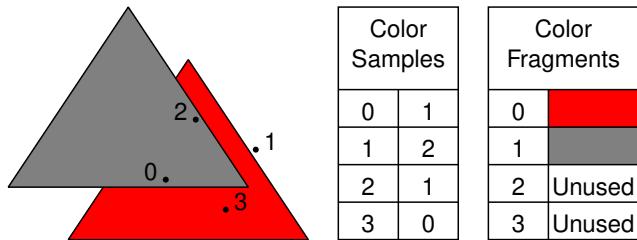


Figure 17. Fragment Mask

## 26.5. Custom Sample Locations

Applications **can** also control the sample locations used for rasterization.

If the `pNext` chain of the `VkPipelineMultisampleStateCreateInfo` structure specified at pipeline creation time includes an instance of the `VkPipelineSampleLocationsStateCreateInfoEXT` structure, then that structure controls the sample locations used when rasterizing primitives with the pipeline.

The `VkPipelineSampleLocationsStateCreateInfoEXT` structure is defined as:

```
typedef struct VkPipelineSampleLocationsStateCreateInfoEXT {
    VkStructureType          sType;
    const void*               pNext;
    VkBool32                  sampleLocationsEnable;
    VkSampleLocationsInfoEXT   sampleLocationsInfo;
} VkPipelineSampleLocationsStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `sampleLocationsEnable` controls whether custom sample locations are used. If `sampleLocationsEnable` is `VK_FALSE`, the default sample locations are used and the values specified in `sampleLocationsInfo` are ignored.
- `sampleLocationsInfo` is the sample locations to use during rasterization if `sampleLocationsEnable` is `VK_TRUE` and the graphics pipeline is not created with `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT`
- `sampleLocationsInfo` **must** be a valid `VkSampleLocationsInfoEXT` structure

The `VkSampleLocationsInfoEXT` structure is defined as:

```

typedef struct VkSampleLocationsInfoEXT {
    VkStructureType          sType;
    const void*             pNext;
    VkSampleCountFlagBits    sampleLocationsPerPixel;
    VkExtent2D                sampleLocationGridSize;
    uint32_t                  sampleLocationsCount;
    const VkSampleLocationEXT* pSampleLocations;
} VkSampleLocationsInfoEXT;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `sampleLocationsPerPixel` is a `VkSampleCountFlagBits` specifying the number of sample locations per pixel.
- `sampleLocationGridSize` is the size of the sample location grid to select custom sample locations for.
- `sampleLocationsCount` is the number of sample locations in `pSampleLocations`.
- `pSampleLocations` is a pointer to an array of `sampleLocationsCount` `VkSampleLocationEXT` structures.

This structure **can** be used either to specify the sample locations to be used for rendering or to specify the set of sample locations an image subresource has been last rendered with for the purposes of layout transitions of depth/stencil images created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT`.

The sample locations in `pSampleLocations` specify `sampleLocationsPerPixel` number of sample locations for each pixel in the grid of the size specified in `sampleLocationGridSize`. The sample location for sample  $i$  at the pixel grid location  $(x,y)$  is taken from `pSampleLocations[(x + y * sampleLocationGridSize.width) * sampleLocationsPerPixel + i]`.

If the render pass has a fragment density map, the implementation will choose the sample locations for the fragment and the contents of `pSampleLocations` **may** be ignored.

## Valid Usage

- `sampleLocationsPerPixel` **must** be a bit value that is set in `VkPhysicalDeviceSampleLocationsPropertiesEXT::sampleLocationSampleCounts`
- `sampleLocationsCount` **must** equal `sampleLocationsPerPixel × sampleLocationGridSize.width × sampleLocationGridSize.height`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT`
- If `sampleLocationsPerPixel` is not `0`, `sampleLocationsPerPixel` must be a valid `VkSampleCountFlagBits` value
- If `sampleLocationsCount` is not `0`, `pSampleLocations` must be a valid pointer to an array of `sampleLocationsCount` `VkSampleLocationEXT` structures

The `VkSampleLocationEXT` structure is defined as:

```
typedef struct VkSampleLocationEXT {
    float    x;
    float    y;
} VkSampleLocationEXT;
```

- `x` is the horizontal coordinate of the sample's location.
- `y` is the vertical coordinate of the sample's location.

The domain space of the sample location coordinates has an upper-left origin within the pixel in framebuffer space.

The values specified in a `VkSampleLocationEXT` structure are always clamped to the implementation-dependent sample location coordinate range `[sampleLocationCoordinateRange[0],sampleLocationCoordinateRange[1]]` that can be queried by chaining the `VkPhysicalDeviceSampleLocationsPropertiesEXT` structure to the `pNext` chain of `VkPhysicalDeviceProperties2`.

The custom sample locations used for rasterization when `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` is `VK_TRUE` are specified by the `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsInfo` property of the bound graphics pipeline, if the pipeline was not created with `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT` enabled.

Otherwise, the sample locations used for rasterization are set by calling `vkCmdSetSampleLocationsEXT`:

```
void vkCmdSetSampleLocationsEXT(
    VkCommandBuffer                                commandBuffer,
    const VkSampleLocationsInfoEXT*                pSampleLocationsInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pSampleLocationsInfo` is the sample locations state to set.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT` dynamic state enabled
- The `sampleLocationsPerPixel` member of `pSampleLocationsInfo` **must** equal the `rasterizationSamples` member of the `VkPipelineMultisampleStateCreateInfo` structure the bound graphics pipeline has been created with
- If `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE` then the current render pass **must** have been begun by specifying a `VkRenderPassSampleLocationsBeginInfoEXT` structure whose `pPostSubpassSampleLocations` member contains an element with a `subpassIndex` matching the current subpass index and the `sampleLocationsInfo` member of that element **must** match the sample locations state pointed to by `pSampleLocationsInfo`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pSampleLocationsInfo` **must** be a valid pointer to a valid `VkSampleLocationsInfoEXT` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

## 26.6. Shading Rate Image

The `shading rate image` feature allows pipelines to use a `shading rate image` to control the `fragment area` and the minimum number of fragment shader invocations launched for each fragment. When

the shading rate image is enabled, the rasterizer determines a base [shading rate](#) for each region of the framebuffer covered by a primitive by fetching a value from the shading rate image and translating it to a shading rate using a per-viewport shading rate palette. This base shading rate is then adjusted to derive a final shading rate. The final shading rate specifies the fragment area and fragment shader invocation count to use for fragments generated in the region.

If the `pNext` chain of [VkPipelineViewportStateCreateInfo](#) includes a [VkPipelineViewportShadingRateImageStateCreateInfoNV](#) structure, then that structure includes parameters that control the shading rate.

The [VkPipelineViewportShadingRateImageStateCreateInfoNV](#) structure is defined as:

```
typedef struct VkPipelineViewportShadingRateImageStateCreateInfoNV {
    VkStructureType           sType;
    const void*               pNext;
    VkBool32                  shadingRateImageEnable;
    uint32_t                  viewportCount;
    const VkShadingRatePaletteNV*   pShadingRatePalettes;
} VkPipelineViewportShadingRateImageStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shadingRateImageEnable` specifies whether shading rate image and palettes are used during rasterization.
- `viewportCount` specifies the number of per-viewport palettes used to translate values stored in shading rate images.
- `pShadingRatePalettes` is a pointer to an array of [VkShadingRatePaletteNV](#) structures defining the palette for each viewport. If the shading rate palette state is dynamic, this member is ignored.

If this structure is not present, `shadingRateImageEnable` is considered to be `VK_FALSE`, and the shading rate image and palettes are not used.

## Valid Usage

- If the [multiple viewports](#) feature is not enabled, `viewportCount` **must** be `0` or `1`
- `viewportCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxViewports`
- If `shadingRateImageEnable` is `VK_TRUE`, `viewportCount` **must** be equal to the `viewportCount` member of [VkPipelineViewportStateCreateInfo](#)
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV`, `pShadingRatePalettes` **must** be a valid pointer to an array of `viewportCount` [VkShadingRatePaletteNV](#) structures

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SHADING_RATE_IMAGE_STATE_CREATE_INFO_NV`
- If `viewportCount` is not `0`, and `pShadingRatePalettes` is not `NULL`, `pShadingRatePalettes` must be a valid pointer to an array of `viewportCount` valid `VkShadingRatePaletteNV` structures

When shading rate image usage is enabled in the bound pipeline, the pipeline uses a shading rate image specified by the command:

```
void vkCmdBindShadingRateImageNV(  
    VkCommandBuffer  
        commandBuffer,  
    VkImageView  
        imageView,  
    VkImageLayout  
        imageLayout);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `imageView` is an image view handle specifying the shading rate image. `imageView` may be set to `VK_NULL_HANDLE`, which is equivalent to specifying a view of an image filled with zero values.
- `imageLayout` is the layout that the image subresources accessible from `imageView` will be in when the shading rate image is accessed.

## Valid Usage

- The `shading rate image` feature must be enabled.
- If `imageView` is not `VK_NULL_HANDLE`, it must be a valid `VkImageView` handle of type `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`.
- If `imageView` is not `VK_NULL_HANDLE`, it must have a format of `VK_FORMAT_R8_UINT`.
- If `imageView` is not `VK_NULL_HANDLE`, it must have been created with a `usage` value including `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`
- If `imageView` is not `VK_NULL_HANDLE`, `imageLayout` must match the actual `VkImageLayout` of each subresource accessible from `imageView` at the time the subresource is accessed.
- If `imageView` is not `VK_NULL_HANDLE`, `imageLayout` must be `VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV` or `VK_IMAGE_LAYOUT_GENERAL`.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- If `imageView` is not `VK_NULL_HANDLE`, `imageView` **must** be a valid `VkImageView` handle
- `imageLayout` **must** be a valid `VkImageLayout` value
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- Both of `commandBuffer`, and `imageView` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

When the shading rate image is enabled in the current pipeline, rasterizing a primitive covering the pixel with coordinates  $(x,y)$  will fetch a shading rate index value from the shading rate image bound by `vkCmdBindShadingRateImageNV`. If the shading rate image view has a type of `VK_IMAGE_VIEW_TYPE_2D`, the lookup will use texel coordinates  $(u,v)$  where  $u = \lfloor \frac{x}{twidht} \rfloor$ ,  $v = \lfloor \frac{y}{theight} \rfloor$ , and `twidht` and `theight` are the width and height of the implementation-dependent `shading rate texel size`. If the shading rate image view has a type of `VK_IMAGE_VIEW_TYPE_2D_ARRAY`, the lookup will use texel coordinates  $(u,v)$  to extract a texel from the layer  $l$ , where  $l$  is the layer of the framebuffer being rendered to. If  $l$  is greater than or equal to the number of layers in the image view, layer zero will be used.

If the bound shading rate image view is not `VK_NULL_HANDLE` and contains a texel with coordinates  $(u,v)$  in layer  $l$  (if applicable), the single unsigned integer component for that texel will be used as the shading rate index. If the  $(u,v)$  coordinate is outside the extents of the subresource used by the shading rate image view, or if the image view is `VK_NULL_HANDLE`, the shading rate index is zero. If the shading rate image view has multiple mipmap levels, the base level identified by `VkImageSubresourceRange::baseMipLevel` will be used.

A shading rate index is mapped to a base shading rate using a lookup table called the shading rate

image palette. There is a separate palette for each viewport. The number of entries in each palette is given by the implementation-dependent [shading rate image palette size](#).

If a pipeline state object is created with `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` enabled, the per-viewport shading rate image palettes are set by the command:

```
void vkCmdSetViewportShadingRatePaletteNV(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkShadingRatePaletteNV*  
                                commandBuffer,  
                                firstViewport,  
                                viewportCount,  
                                pShadingRatePalettes);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstViewport` is the index of the first viewport whose shading rate palette is updated by the command.
- `viewportCount` is the number of viewports whose shading rate palettes are updated by the command.
- `pShadingRatePalettes` is a pointer to an array of `VkShadingRatePaletteNV` structures defining the palette for each viewport.

## Valid Usage

- The [shading rate image](#) feature **must** be enabled.
- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled
- `firstViewport` **must** be less than `VkPhysicalDeviceLimits::maxViewports`
- The sum of `firstViewport` and `viewportCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- If the [multiple viewports](#) feature is not enabled, `firstViewport` **must** be 0
- If the [multiple viewports](#) feature is not enabled, `viewportCount` **must** be 1

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pShadingRatePalettes` **must** be a valid pointer to an array of `viewportCount` valid `VkShadingRatePaletteNV` structures
- `commandBuffer` **must** be in the [recording state](#)
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `viewportCount` **must** be greater than 0

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

The `VkShadingRatePaletteNV` structure specifies to contents of a single shading rate image palette and is defined as:

```
typedef struct VkShadingRatePaletteNV {
    uint32_t           shadingRatePaletteEntryCount;
    const VkShadingRatePaletteEntryNV* pShadingRatePaletteEntries;
} VkShadingRatePaletteNV;
```

- `shadingRatePaletteEntryCount` specifies the number of entries in the shading rate image palette.
- `pShadingRatePaletteEntries` is a pointer to an array of `VkShadingRatePaletteEntryNV` enums defining the shading rate for each palette entry.

## Valid Usage

- `shadingRatePaletteEntryCount` **must** be between 1 and `VkPhysicalDeviceShadingRateImagePropertiesNV::shadingRatePaletteSize`, inclusive

## Valid Usage (Implicit)

- `pShadingRatePaletteEntries` **must** be a valid pointer to an array of `shadingRatePaletteEntryCount` valid `VkShadingRatePaletteEntryNV` values
- `shadingRatePaletteEntryCount` **must** be greater than 0

To determine the base shading rate image, a shading rate index  $i$  is mapped to array element  $i$  in the array `pShadingRatePaletteEntries` for the palette corresponding to the viewport used for the fragment. If  $i$  is greater than or equal to the palette size `shadingRatePaletteEntryCount`, the base shading rate is undefined.

The supported shading rate image palette entries are defined by `VkShadingRatePaletteEntryNV`:

```

typedef enum VkShadingRatePaletteEntryNV {
    VK_SHADING_RATE_PALETTE_ENTRY_NO_INVOCATIONS_NV = 0,
    VK_SHADING_RATE_PALETTE_ENTRY_16_INVOCATIONS_PER_PIXEL_NV = 1,
    VK_SHADING_RATE_PALETTE_ENTRY_8_INVOCATIONS_PER_PIXEL_NV = 2,
    VK_SHADING_RATE_PALETTE_ENTRY_4_INVOCATIONS_PER_PIXEL_NV = 3,
    VK_SHADING_RATE_PALETTE_ENTRY_2_INVOCATIONS_PER_PIXEL_NV = 4,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_PIXEL_NV = 5,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X1_PIXELS_NV = 6,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_1X2_PIXELS_NV = 7,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X2_PIXELS_NV = 8,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X2_PIXELS_NV = 9,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X4_PIXELS_NV = 10,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X4_PIXELS_NV = 11,
    VK_SHADING_RATE_PALETTE_ENTRY_MAX_ENUM_NV = 0x7FFFFFFF
} VkShadingRatePaletteEntryNV;

```

The following table indicates the width and height (in pixels) of each fragment generated using the indicated shading rate, as well as the maximum number of fragment shader invocations launched for each fragment. When processing regions of a primitive that have a shading rate of `VK_SHADING_RATE_PALETTE_ENTRY_NO_INVOCATIONS_NV`, no fragments will be generated in that region.

<b>Shading Rate</b>	<b>Width</b>	<b>Height</b>	<b>Invocations</b>
<code>VK_SHADING_RATE_PALETTE_ENTRY_NO_INVOCATIONS_NV</code>	0	0	0
<code>VK_SHADING_RATE_PALETTE_ENTRY_16_INVOCATIONS_PER_PIXEL_NV</code>	1	1	16
<code>VK_SHADING_RATE_PALETTE_ENTRY_8_INVOCATIONS_PER_PIXEL_NV</code>	1	1	8
<code>VK_SHADING_RATE_PALETTE_ENTRY_4_INVOCATIONS_PER_PIXEL_NV</code>	1	1	4
<code>VK_SHADING_RATE_PALETTE_ENTRY_2_INVOCATIONS_PER_PIXEL_NV</code>	1	1	2
<code>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X1_PIXELS_NV</code>	1	1	1
<code>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_1X2_PIXELS_NV</code>	2	1	1
<code>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X2_PIXELS_NV</code>	1	2	1
<code>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X2_PIXELS_NV</code>	2	2	1
<code>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X4_PIXELS_NV</code>	4	2	1

Shading Rate	Width	Height	Invocations
VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X4_PIXELS_NV	2	4	1
VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X4_PIXELS_NV	4	4	1

When the shading rate image is disabled, a shading rate of `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_PIXEL_NV` will be used as the base shading rate.

Once a base shading rate has been established, it is adjusted to produce a final shading rate. First, if the base shading rate uses multiple pixels for each fragment, the implementation **may** reduce the fragment area to ensure that the total number of coverage samples for all pixels in a fragment does not exceed [an implementation-dependent maximum](#).

If [sample shading](#) is active in the current pipeline and would result in processing  $n$  ( $n > 1$ ) unique samples per fragment when the shading rate image is disabled, the shading rate is adjusted in an implementation-dependent manner to increase the number of fragment shader invocations spawned by the primitive. If the shading rate indicates  $fs$  pixels per fragment and  $fs$  is greater than  $n$ , the fragment area is adjusted so each fragment has approximately  $f_n^s$  pixels. Otherwise, if the shading rate indicates  $ipf$  invocations per fragment, the fragment area will be adjusted to a single pixel with approximately  $ipf \times \frac{n}{f}s$  invocations per fragment.

If sample shading occurs due to the use of a fragment shader input variable decorated with `SampleId` or `SamplePosition`, the shading rate is ignored. Each fragment will have a single pixel and will spawn up to `totalSamples` fragment shader invocations, as when using [sample shading](#) without a shading rate image.

Finally, if the shading rate specifies multiple fragment shader invocations per fragment, the total number of invocations in the shading rate is clamped to be no larger than the value of `totalSamples` used for [sample shading](#).

When the final shading rate for a primitive covering pixel  $(x,y)$  has a fragment area of  $fw \times fh$ , the fragment for that pixel will cover all pixels with coordinates  $(x',y')$  that satisfy the equations:

$$\lfloor \frac{x}{fw} \rfloor = \lfloor \frac{x'}{fw} \rfloor$$

$$\lfloor \frac{y}{fh} \rfloor = \lfloor \frac{y'}{fh} \rfloor$$

This combined fragment is considered to have multiple coverage samples; the total number of samples in this fragment is given by  $samples = fw \times fh \times rs$  where  $rs$  indicates the value of `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` specified at pipeline creation time. The set of coverage samples in the fragment is the union of the per-pixel coverage samples in each of the fragment's pixels. The location and order of coverage samples within each pixel in the combined fragment are assigned as described in [Multisampling](#) and [Custom Sample Locations](#). Each coverage sample in the set of pixels belonging to the combined fragment is assigned a unique sample number in the range  $[0, samples - 1]$ . If the `shadingRateCoarseSampleOrder` feature is supported, the order of coverage samples **can** be specified for each combination of fragment area and coverage

sample count. If this feature is not supported, the sample order is implementation-dependent.

If the `pNext` chain of `VkPipelineViewportStateCreateInfo` includes a `VkPipelineViewportCoarseSampleOrderStateCreateInfoNV` structure, then that structure includes parameters that control the order of coverage samples in fragments larger than one pixel.

The `VkPipelineViewportCoarseSampleOrderStateCreateInfoNV` structure is defined as:

```
typedef struct VkPipelineViewportCoarseSampleOrderStateCreateInfoNV {
    VkStructureType           sType;
    const void*               pNext;
    VkCoarseSampleOrderTypeNV sampleOrderType;
    uint32_t                  customSampleOrderCount;
    const VkCoarseSampleOrderCustomNV* pCustomSampleOrders;
} VkPipelineViewportCoarseSampleOrderStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `sampleOrderType` specifies the mechanism used to order coverage samples in fragments larger than one pixel.
- `customSampleOrderCount` specifies the number of custom sample orderings to use when ordering coverage samples.
- `pCustomSampleOrders` is a pointer to an array of `customSampleOrderCount` `VkCoarseSampleOrderCustomNV` structures, each of which specifies the coverage sample order for a single combination of fragment area and coverage sample count.

If this structure is not present, `sampleOrderType` is considered to be `VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV`.

If `sampleOrderType` is `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV`, the coverage sample order used for any combination of fragment area and coverage sample count not enumerated in `pCustomSampleOrders` will be identical to that used for `VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV`.

If the pipeline was created with `VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV`, the contents of this structure (if present) are ignored, and the coverage sample order is instead specified by `vkCmdSetCoarseSampleOrderNV`.

## Valid Usage

- If `sampleOrderType` is not `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV`, `customSampleOrderCount` **must** be 0
- The array `pCustomSampleOrders` **must** not contain two structures with matching values for both the `shadingRate` and `sampleCount` members.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_COARSE_SAMPLE_ORDER_STATE_CREATE_INFO_NV`
- `sampleOrderType` must be a valid `VkCoarseSampleOrderTypeNV` value
- If `customSampleOrderCount` is not `0`, `pCustomSampleOrders` must be a valid pointer to an array of `customSampleOrderCount` valid `VkCoarseSampleOrderCustomNV` structures

The type `VkCoarseSampleOrderTypeNV` specifies the technique used to order coverage samples in fragments larger than one pixel, and is defined as:

```
typedef enum VkCoarseSampleOrderTypeNV {  
    VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV = 0,  
    VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV = 1,  
    VK_COARSE_SAMPLE_ORDER_TYPE_PIXEL_MAJOR_NV = 2,  
    VK_COARSE_SAMPLE_ORDER_TYPE_SAMPLE_MAJOR_NV = 3,  
    VK_COARSE_SAMPLE_ORDER_TYPE_MAX_ENUM_NV = 0x7FFFFFFF  
} VkCoarseSampleOrderTypeNV;
```

- `VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV` specifies that coverage samples will be ordered in an implementation-dependent manner.
- `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV` specifies that coverage samples will be ordered according to the array of custom orderings provided in either the `pCustomSampleOrders` member of `VkPipelineViewportCoarseSampleOrderStateCreateInfoNV` or the `pCustomSampleOrders` member of `vkCmdSetCoarseSampleOrderNV`.
- `VK_COARSE_SAMPLE_ORDER_TYPE_PIXEL_MAJOR_NV` specifies that coverage samples will be ordered sequentially, sorted first by pixel coordinate (in row-major order) and then by coverage sample number.
- `VK_COARSE_SAMPLE_ORDER_TYPE_SAMPLE_MAJOR_NV` specifies that coverage samples will be ordered sequentially, sorted first by coverage sample number and then by pixel coordinate (in row-major order).

When using a coarse sample order of `VK_COARSE_SAMPLE_ORDER_TYPE_PIXEL_MAJOR_NV` for a fragment with an upper-left corner of  $(fx, fy)$  with a width of  $fw \times fh$  and  $fsc$  coverage samples per pixel, sample  $cs$  of the fragment will be assigned to sample  $fs$  of pixel  $(px, py)$  will be assigned as follows:

$$\begin{aligned} px &= fx + (\lfloor c \frac{s}{f} sc \rfloor \% fw) \\ py &= fy + \lfloor c \frac{s}{fsc \times fw} \rfloor \\ fs &= cs \% fsc \end{aligned}$$

When using a coarse sample order of `VK_COARSE_SAMPLE_ORDER_TYPE_SAMPLE_MAJOR_NV`, sample  $cs$  will be assigned as follows:

$$\begin{aligned}
 px &= fx + cs\%fw \\
 py &= (fy + \lfloor c \frac{s}{f} w \rfloor \%fh) \\
 fs &= \lfloor c \frac{s}{fw \times fh} \rfloor
 \end{aligned}$$

The `VkCoarseSampleOrderCustomNV` structure is used with a coverage sample ordering type of `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV` to specify the order of coverage samples for one combination of fragment width, fragment height, and coverage sample count. The structure is defined as:

```

typedef struct VkCoarseSampleOrderCustomNV {
    VkShadingRatePaletteEntryNV      shadingRate;
    uint32_t                      sampleCount;
    uint32_t                      sampleLocationCount;
    const VkCoarseSampleLocationNV* pSampleLocations;
} VkCoarseSampleOrderCustomNV;

```

- `shadingRate` is a shading rate palette entry that identifies the fragment width and height for the combination of fragment area and per-pixel coverage sample count to control.
- `sampleCount` identifies the per-pixel coverage sample count for the combination of fragment area and coverage sample count to control.
- `sampleLocationCount` specifies the number of sample locations in the custom ordering.
- `pSampleLocations` is a pointer to an array of `VkCoarseSampleOrderCustomNV` structures specifying the location of each sample in the custom ordering.

When using a custom sample ordering, element  $i$  in `pSampleLocations` specifies a specific pixel and per-pixel coverage sample number that corresponds to the coverage sample numbered  $i$  in the multi-pixel fragment.

## Valid Usage

- `shadingRate` **must** be a shading rate that generates fragments with more than one pixel.
- `sampleCount` **must** correspond to a sample count enumerated in `VkSampleCountFlags` whose corresponding bit is set in `VkPhysicalDeviceLimits::framebufferNoAttachmentsSampleCounts`.
- `sampleLocationCount` **must** be equal to the product of `sampleCount`, the fragment width for `shadingRate`, and the fragment height for `shadingRate`.
- `sampleLocationCount` **must** be less than or equal to the value of `VkPhysicalDeviceShadingRateImagePropertiesNV::shadingRateMaxCoarseSamples`.
- The array `pSampleLocations` **must** contain exactly one entry for every combination of valid values for `pixelX`, `pixelY`, and `sample` in the structure `VkCoarseSampleOrderCustomNV`.

## Valid Usage (Implicit)

- `shadingRate` **must** be a valid `VkShadingRatePaletteEntryNV` value
- `pSampleLocations` **must** be a valid pointer to an array of `sampleLocationCount` `VkCoarseSampleLocationNV` structures
- `sampleLocationCount` **must** be greater than 0

The `VkCoarseSampleLocationNV` structure identifies a specific pixel and sample number for one of the coverage samples in a fragment that is larger than one pixel. This structure is defined as:

```
typedef struct VkCoarseSampleLocationNV {  
    uint32_t    pixelX;  
    uint32_t    pixelY;  
    uint32_t    sample;  
} VkCoarseSampleLocationNV;
```

- `pixelX` is added to the x coordinate of the upper-leftmost pixel of each fragment to identify the pixel containing the coverage sample.
- `pixelY` is added to the y coordinate of the upper-leftmost pixel of each fragment to identify the pixel containing the coverage sample.
- `sample` is the number of the coverage sample in the pixel identified by `pixelX` and `pixelY`.

## Valid Usage

- `pixelX` **must** be less than the width (in pixels) of the fragment.
- `pixelY` **must** be less than the height (in pixels) of the fragment.
- `sample` **must** be less than the number of coverage samples in each pixel belonging to the fragment.

If a pipeline state object is created with `VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV` enabled, the order of coverage samples in fragments larger than one pixel is set by the command:

```
void vkCmdSetCoarseSampleOrderNV(  
    VkCommandBuffer                      commandBuffer,  
    VkCoarseSampleOrderTypeNV            sampleOrderType,  
    uint32_t                            customSampleOrderCount,  
    const VkCoarseSampleOrderCustomNV*  pCustomSampleOrders);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `sampleOrderType` specifies the mechanism used to order coverage samples in fragments larger than one pixel.

- `customSampleOrderCount` specifies the number of custom sample orderings to use when ordering coverage samples.
- `pCustomSampleOrders` is a pointer to an array of `VkCoarseSampleOrderCustomNV` structures, each of which specifies the coverage sample order for a single combination of fragment area and coverage sample count.

If `sampleOrderType` is `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV`, the coverage sample order used for any combination of fragment area and coverage sample count not enumerated in `pCustomSampleOrders` will be identical to that used for `VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV`.

## Valid Usage

- If `sampleOrderType` is not `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV`, `customSampleOrderCount` **must** be `0`
- The array `pCustomSampleOrders` **must** not contain two structures with matching values for both the `shadingRate` and `sampleCount` members.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `sampleOrderType` **must** be a valid `VkCoarseSampleOrderTypeNV` value
- If `customSampleOrderCount` is not `0`, `pCustomSampleOrders` **must** be a valid pointer to an array of `customSampleOrderCount` valid `VkCoarseSampleOrderCustomNV` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

If the final shading rate for a primitive covering pixel  $(x,y)$  results in  $n$  invocations per pixel ( $n > 1$ ),

$n$  separate fragment shader invocations will be generated for the fragment. Each coverage sample in the fragment will be assigned to one of the  $n$  fragment shader invocations in an implementation-dependent manner. The outputs from the [fragment output interface](#) of each shader invocation will be broadcast to all of the framebuffer samples associated with the invocation. If none of the coverage samples associated with a fragment shader invocation is covered by a primitive, the implementation **may** discard the fragment shader invocation for those samples.

If the final shading rate for a primitive covering pixel  $(x,y)$  results in a fragment containing multiple pixels, a single set of fragment shader invocations will be generated for all pixels in the combined fragment. Outputs from the [fragment output interface](#) will be broadcast to all covered framebuffer samples belonging to the fragment. If the fragment shader executes code discarding the fragment, none of the samples of the fragment will be updated.

## 26.7. Sample Shading

Sample shading **can** be used to specify a minimum number of unique samples to process for each fragment. If sample shading is enabled an implementation **must** provide a minimum of  $\text{max}(\text{minSampleShadingFactor} \times \text{totalSamples}, 1)$  unique associated data for each fragment, where `minSampleShadingFactor` is the minimum fraction of sample shading. If the `VK_AMD_mixed_attachment_samples` extension is enabled and the subpass uses color attachments, `totalSamples` is the number of samples of the color attachments. Otherwise, `totalSamples` is the value of `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` specified at pipeline creation time. These are associated with the samples in an implementation-dependent manner. When `minSampleShadingFactor` is `1.0`, a separate set of associated data are evaluated for each sample, and each set of values is evaluated at the sample location.

Sample shading is enabled for a graphics pipeline:

- If the interface of the fragment shader entry point of the graphics pipeline includes an input variable decorated with `SampleId` or `SamplePosition`. In this case `minSampleShadingFactor` takes the value `1.0`.
- Else if the `sampleShadingEnable` member of the `VkPipelineMultisampleStateCreateInfo` structure specified when creating the graphics pipeline is set to `VK_TRUE`. In this case `minSampleShadingFactor` takes the value of `VkPipelineMultisampleStateCreateInfo::minSampleShading`.

Otherwise, sample shading is considered disabled.

## 26.8. Barycentric Interpolation

When the `fragmentShaderBarycentric` feature is enabled, the [PerVertexNV interpolation decoration](#) **can** be used with fragment shader inputs to indicate that the decorated inputs do not have associated data in the fragment. Such inputs **can** only be accessed in a fragment shader using an array index whose value (0, 1, or 2) identifies one of the vertices of the primitive that produced the fragment.

When [tessellation](#), [geometry shading](#), and [mesh shading](#) are not active, fragment shader inputs decorated with `PerVertexNV` will take values from one of the vertices of the primitive that produced

the fragment, identified by the extra index provided in SPIR-V code accessing the input. If the  $n$  vertices passed to a draw call are numbered 0 through  $n-1$ , and the point, line, and triangle primitives produced by the draw call are numbered with consecutive integers beginning with zero, the following table indicates the original vertex numbers used for index values of 0, 1, and 2. If an input decorated with `PerVertexNV` is accessed with any other vertex index value, the value obtained is undefined.

Primitive Topology	Vertex 0	Vertex 1	Vertex 2
<code>VK_PRIMITIVE_TOPOLOGY_POINT_LIST</code>	i	-	-
<code>VK_PRIMITIVE_TOPOLOGY_LINE_LIST</code>	2i	2i+1	-
<code>VK_PRIMITIVE_TOPOLOGY_LINE_STRIP</code>	i	i+1	-
<code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST</code>	3i	3i+1	3i+2
<code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP</code> (even)	i	i+1	i+2
<code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP</code> (odd)	i	i+2	i+1
<code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN</code>	i+1	i+2	0
<code>VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY</code>	4i+1	4i+2	-
<code>VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY</code>	i+1	i+2	-
<code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY</code>	6i	6i+2	6i+4
<code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY</code> (even)	2i	2i+2	2i+4
<code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY</code> (odd)	2i	2i+4	2i+2

When geometry or mesh shading is active, primitives processed by fragment shaders are assembled from the vertices emitted by the geometry or mesh shader. In this case, the vertices used for fragment shader inputs decorated with `PerVertexNV` are derived by treating the primitives produced by the shader as though they were specified by a draw call and consulting [the table above](#).

When using tessellation without geometry shading, the tessellator produces primitives in an implementation-dependent manner. While there is no defined vertex ordering for inputs decorated with `PerVertexNV`, the vertex ordering used in this case will be consistent with the ordering used to derive the values of inputs decorated with `BaryCoordNV` or `BaryCoordNoPerspNV`.

Fragment shader inputs decorated with `BaryCoordNV` or `BaryCoordNoPerspNV` hold three-component vectors with barycentric weights that indicate the location of the fragment relative to the screen-space locations of vertices of its primitive. For point primitives, such variables are always assigned

the value (1,0,0). For `line` primitives, the built-ins are obtained by interpolating an attribute whose values for the vertices numbered 0 and 1 are (1,0,0) and (0,1,0), respectively. For `polygon` primitives, the built-ins are obtained by interpolating an attribute whose values for the vertices numbered 0, 1, and 2 are (1,0,0), (0,1,0), and (0,0,1), respectively. For `BaryCoordNV`, the values are obtained using perspective interpolation. For `BaryCoordNoPerspNV`, the values are obtained using linear interpolation.

## 26.9. Points

A point is drawn by generating a set of fragments in the shape of a square centered around the vertex of the point. Each vertex has an associated point size that controls the width/height of that square. The point size is taken from the (potentially clipped) shader built-in `PointSize` written by:

- the geometry shader, if active;
- the tessellation evaluation shader, if active and no geometry shader is active;
- the vertex shader, otherwise

and clamped to the implementation-dependent point size range `[pointSizeRange[0], pointSizeRange[1]]`. The value written to `PointSize` **must** be greater than zero.

Not all point sizes need be supported, but the size 1.0 **must** be supported. The range of supported sizes and the size of evenly-spaced gradations within that range are implementation-dependent. The range and gradations are obtained from the `pointSizeRange` and `pointSizeGranularity` members of `VkPhysicalDeviceLimits`. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the size 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional point sizes **may** also be supported. There is no requirement that these sizes be equally spaced. If an unsupported size is requested, the nearest supported size is used instead.

Further, if the render pass has a fragment density map attachment, point size **may** be rounded by the implementation to a multiple of the fragment's width or height.

### 26.9.1. Basic Point Rasterization

Point rasterization produces a fragment for each fragment area group of framebuffer pixels with one or more sample points that intersect a region centered at the point's  $(x_p, y_p)$ . This region is a square with side equal to the current point size. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All fragments produced in rasterizing a point are assigned the same associated data, which are those of the vertex corresponding to the point. However, the fragment shader built-in `PointCoord` contains point sprite texture coordinates. The s and t point sprite texture coordinates vary from zero to one across the point horizontally left-to-right and top-to-bottom, respectively. The following formulas are used to evaluate s and t:

$$s = \frac{1}{2} + \frac{(x_p - x_f)}{\text{size}}$$

$$t = \frac{1}{2} + \frac{(y_p - y_f)}{\text{size}}$$

where size is the point's size;  $(x_p, y_p)$  is the location at which the point sprite coordinates are

evaluated - this **may** be the framebuffer coordinates of the fragment center, or the location of a sample; and  $(x_f, y_f)$  is the exact, unrounded framebuffer coordinate of the vertex for the point.

## 26.10. Line Segments

Line segment rasterization options are controlled by the [VkPipelineRasterizationLineStateCreateInfoEXT](#) structure.

The [VkPipelineRasterizationLineStateCreateInfoEXT](#) structure is defined as:

```
typedef struct VkPipelineRasterizationLineStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkLineRasterizationModeEXT lineRasterizationMode;
    VkBool32 stippledLineEnable;
    uint32_t lineStippleFactor;
    uint16_t lineStipplePattern;
} VkPipelineRasterizationLineStateCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **lineRasterizationMode** is a [VkLineRasterizationModeEXT](#) value selecting the style of line rasterization.
- **stippledLineEnable** enables [stippled line rasterization](#).
- **lineStippleFactor** is the repeat factor used in stippled line rasterization.
- **lineStipplePattern** is the bit pattern used in stippled line rasterization.

## Valid Usage

- If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT`, then the `rectangularLines` feature **must** be enabled
- If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT`, then the `bresenhamLines` feature **must** be enabled
- If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, then the `smoothLines` feature **must** be enabled
- If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT`, then the `stippledRectangularLines` feature **must** be enabled
- If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT`, then the `stippledBresenhamLines` feature **must** be enabled
- If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, then the `stippledSmoothLines` feature **must** be enabled
- If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT`, then the `stippledRectangularLines` feature **must** be enabled and `VkPhysicalDeviceLimits::strictLines` **must** be `VK_TRUE`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT`
- `lineRasterizationMode` **must** be a valid `VkLineRasterizationModeEXT` value

Possible values of `VkPipelineRasterizationLineStateCreateInfoEXT::lineRasterizationMode` are:

```
typedef enum VkLineRasterizationModeEXT {
    VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT = 0,
    VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT = 1,
    VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT = 2,
    VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT = 3,
    VK_LINE_RASTERIZATION_MODE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkLineRasterizationModeEXT;
```

- `VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT` is equivalent to `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` if `VkPhysicalDeviceLimits::strictLines` is `VK_TRUE`, otherwise lines are drawn as non-`strictLines` parallelograms. Both of these modes are defined in [Basic Line Segment Rasterization](#).
- `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` specifies lines drawn as if they were rectangles extruded from the line

- `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` specifies lines drawn by determining which pixel diamonds the line intersects and exits, as defined in [Bresenham Line Segment Rasterization](#).
- `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` specifies lines drawn if they were rectangles extruded from the line, with alpha falloff, as defined in [Smooth Lines](#).

Each line segment has an associated width. The line width is specified by the `VkPipelineRasterizationStateCreateInfo::lineWidth` property of the currently active pipeline, if the pipeline was not created with `VK_DYNAMIC_STATE_LINE_WIDTH` enabled.

Otherwise, the line width is set by calling `vkCmdSetLineWidth`:

```
void vkCmdSetLineWidth(
    VkCommandBuffer
    float
        commandBuffer,
        lineWidth);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `lineWidth` is the width of rasterized line segments.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_LINE_WIDTH` dynamic state enabled
- If the `wide lines` feature is not enabled, `lineWidth` **must** be `1.0`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

Not all line widths need be supported for line segment rasterization, but width 1.0 antialiased segments **must** be provided. The range and gradations are obtained from the `lineWidthRange` and `lineWidthGranularity` members of `VkPhysicalDeviceLimits`. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the size 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional line widths **may** also be supported. There is no requirement that these widths be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.

Further, if the render pass has a fragment density map attachment, line width **may** be rounded by the implementation to a multiple of the fragment's width or height.

### 26.10.1. Basic Line Segment Rasterization

If the `lineRasterizationMode` member of `VkPipelineRasterizationLineStateCreateInfoEXT` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT`, rasterized line segments produce fragments which intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment in directions perpendicular to the direction of the line. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage bits that correspond to sample points that intersect the rectangle are 1, other coverage bits are 0.

Next we specify how the data associated with each rasterized fragment are obtained. Let  $\mathbf{p}_r = (x_d, y_d)$  be the framebuffer coordinates at which associated data are evaluated. This **may** be the center of a fragment or the location of a sample within the fragment. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment center **must** be used. Let  $\mathbf{p}_a = (x_a, y_a)$  and  $\mathbf{p}_b = (x_b, y_b)$  be initial and final endpoints of the line segment, respectively. Set

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}$$

(Note that  $t = 0$  at  $\mathbf{p}_a$  and  $t = 1$  at  $\mathbf{p}_b$ . Also note that this calculation projects the vector from  $\mathbf{p}_a$  to  $\mathbf{p}_r$  onto the line, and thus computes the normalized distance of the fragment along the line.)

The value of an associated datum  $f$  for the fragment, whether it be a shader output or the clip w coordinate, **must** be determined using *perspective interpolation*:

$$f = \frac{(1-t)f_a / w_a + t f_b / w_b}{(1-t) / w_a + t / w_b}$$

where  $f_a$  and  $f_b$  are the data associated with the starting and ending endpoints of the segment, respectively;  $w_a$  and  $w_b$  are the clip w coordinates of the starting and ending endpoints of the segments, respectively.

Depth values for lines **must** be determined using *linear interpolation*:

$$z = (1 - t) z_a + t z_b$$

where  $z_a$  and  $z_b$  are the depth values of the starting and ending endpoints of the segment, respectively.

The `NoPerspective` and `Flat` interpolation decorations **can** be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, `perspective interpolation` is performed as described above. When the `NoPerspective` decoration is used, `linear interpolation` is performed in the same fashion as for depth values, as described above. When the `Flat` decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of the `provoking vertex` corresponding to that primitive.

When the `fragmentShaderBarycentric` feature is enabled, the `PerVertexNV` interpolation decoration **can** also be used with fragment shader inputs which indicate that the decorated inputs are not interpolated and **can** only be accessed using an extra array dimension, where the extra index identifies one of the vertices of the primitive that produced the fragment.

The above description documents the preferred method of line rasterization, and **must** be used when the implementation advertises the `strictLines` limit in `VkPhysicalDeviceLimits` as `VK_TRUE`.

When `strictLines` is `VK_FALSE`, the edges of the lines are generated as a parallelogram surrounding the original line. The major axis is chosen by noting the axis in which there is the greatest distance between the line start and end points. If the difference is equal in both directions then the X axis is chosen as the major axis. Edges 2 and 3 are aligned to the minor axis and are centered on the endpoints of the line as in `Non strict lines`, and each is `lineWidth` long. Edges 0 and 1 are parallel to the line and connect the endpoints of edges 2 and 3. Coverage bits that correspond to sample points that intersect the parallelogram are 1, other coverage bits are 0.

Samples that fall exactly on the edge of the parallelogram follow the polygon rasterization rules.

Interpolation occurs as if the parallelogram was decomposed into two triangles where each pair of vertices at each end of the line has identical attributes.

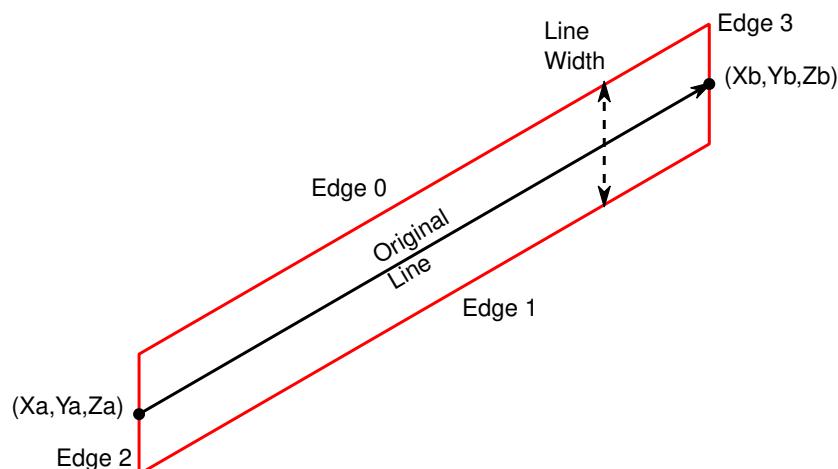


Figure 18. Non strict lines

## 26.10.2. Bresenham Line Segment Rasterization

If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT`, then the following rules replace the line rasterization rules defined in [Basic Line Segment Rasterization](#).

Line segment rasterization begins by characterizing the segment as either *x-major* or *y-major*. *x-major* line segments have slope in the closed interval  $[-1,1]$ ; all other line segments are *y-major* (slope is determined by the segment's endpoints). We specify rasterization only for *x-major* segments except in cases where the modifications for *y-major* segments are not self-evident.

Ideally, Vulkan uses a *diamond-exit* rule to determine those fragments that are produced by rasterizing a line segment. For each fragment  $f$  with center at framebuffer coordinates  $x_f$  and  $y_f$ , define a diamond-shaped region that is the intersection of four half planes:

$$R_f = \{(x, y) \mid |x - x_f| + |y - y_f| < \frac{1}{2}\}$$

Essentially, a line segment starting at  $p_a$  and ending at  $p_b$  produces those fragments  $f$  for which the segment intersects  $R_f$ , except if  $p_b$  is contained in  $R_f$ .

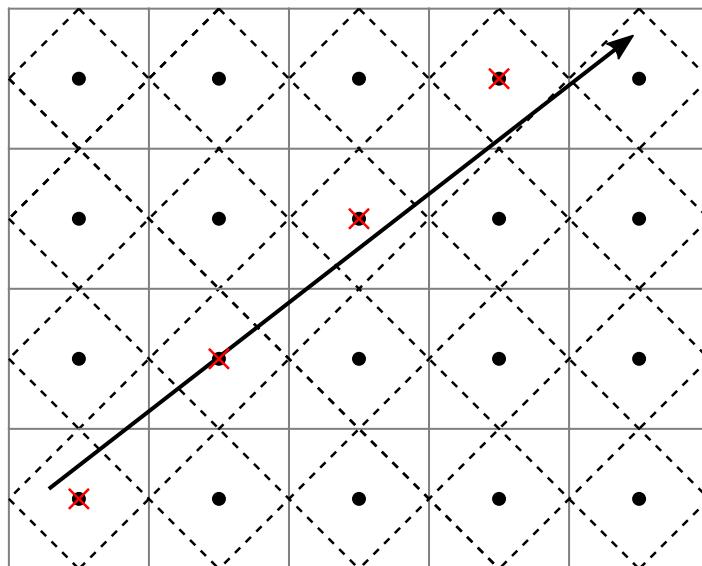


Figure 19. Visualization of Bresenham's algorithm

To avoid difficulties when an endpoint lies on a boundary of  $R_f$  we (in principle) perturb the supplied endpoints by a tiny amount. Let  $p_a$  and  $p_b$  have framebuffer coordinates  $(x_a, y_a)$  and  $(x_b, y_b)$ , respectively. Obtain the perturbed endpoints  $p'_a$  given by  $(x_a, y_a) - (\varepsilon, \varepsilon^2)$  and  $p'_b$  given by  $(x_b, y_b) - (\varepsilon, \varepsilon^2)$ . Rasterizing the line segment starting at  $p_a$  and ending at  $p_b$  produces those fragments  $f$  for which the segment starting at  $p'_a$  and ending on  $p'_b$  intersects  $R_f$ , except if  $p'_b$  is contained in  $R_f$ .  $\varepsilon$  is chosen to be so small that rasterizing the line segment produces the same fragments when  $\delta$  is substituted for  $\varepsilon$  for any  $0 < \delta \leq \varepsilon$ .

When  $p_a$  and  $p_b$  lie on fragment centers, this characterization of fragments reduces to Bresenham's algorithm with one modification: lines produced in this description are "half-open," meaning that the final fragment (corresponding to  $p_b$ ) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham's algorithm).

Implementations **may** use other line segment rasterization algorithms, subject to the following rules:

- The coordinates of a fragment produced by the algorithm **must** not deviate by more than one unit in either x or y framebuffer coordinates from a corresponding fragment produced by the diamond-exit rule.
- The total number of fragments produced by the algorithm **must** not differ from that produced by the diamond-exit rule by no more than one.
- For an x-major line, two fragments that lie in the same framebuffer-coordinate column **must** not be produced (for a y-major line, two fragments that lie in the same framebuffer-coordinate row **must** not be produced).
- If two line segments share a common endpoint, and both segments are either x-major (both left-to-right or both right-to-left) or y-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments **must** not produce duplicate fragments. Fragments also **must** not be omitted so as to interrupt continuity of the connected segments.

The actual width  $w$  of `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` lines is determined by rounding the line width to the nearest integer, clamping it to the implementation-dependent `lineWidthRange` (with both values rounded to the nearest integer), then clamping it to be no less than 1.

`VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` line segments of width other than one are rasterized by offsetting them in the minor direction (for an x-major line, the minor direction is y, and for a y-major line, the minor direction is x) and producing a row or column of fragments in the minor direction. If the line segment has endpoints given by  $(x_0, y_0)$  and  $(x_1, y_1)$  in framebuffer coordinates, the segment with endpoints  $(x_0, y_0 - \frac{w-1}{2})$  and  $(x_1, y_1 - \frac{w-1}{2})$  is rasterized, but instead of a single fragment, a column of fragments of height  $w$  (a row of fragments of length  $w$  for a y-major segment) is produced at each x (y for y-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates.

The preferred method of attribute interpolation for a wide line is to generate the same attribute values for all fragments in the row or column described above, as if the adjusted line were used for interpolation and those values replicated to the other fragments, except for `FragCoord` which is interpolated as usual. Implementations **may** instead interpolate each fragment according to the formula in [Basic Line Segment Rasterization](#), using the original line segment endpoints.

When `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` lines are being rasterized, sample locations **may** all be treated as being at the pixel center (this **may** affect attribute and depth interpolation).

### 26.10.3. Line Stipple

If the `stippledLineEnable` member of `VkPipelineRasterizationLineStateCreateInfoEXT` is `VK_TRUE`, then lines are rasterized with a *line stipple* determined by `lineStippleFactor` and `lineStipplePattern`. `lineStipplePattern` is an unsigned 16-bit integer that determines which fragments are to be drawn or discarded when the line is rasterized. `lineStippleFactor` is a count that is used to modify the effective line stipple by causing each bit in `lineStipplePattern` to be used `lineStippleFactor` times.

Line stippling discards certain fragments that are produced by rasterization. The masking is achieved using three parameters: the 16-bit line stipple pattern  $p$ , the line stipple factor  $r$ , and an integer stipple counter  $s$ . Let

$$b = \lfloor \frac{s}{r} \rfloor \bmod 16$$

Then a fragment is produced if the  $b$ 'th bit of  $p$  is 1, and discarded otherwise. The bits of  $p$  are numbered with 0 being the least significant and 15 being the most significant.

The initial value of  $s$  is zero. For `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` lines,  $s$  is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). For `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` and

`VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` lines, the rectangular region is subdivided into adjacent unit-length rectangles, and  $s$  is incremented once for each rectangle. Rectangles with a value of  $s$  such that the  $b$ 'th bit of  $p$  is zero are discarded. If the last rectangle in a line segment is shorter than unit-length, then the remainder **may** carry over to the next line segment in the line strip using the same value of  $s$  (this is the preferred behavior, for the stipple pattern to appear more consistent through the strip).

$s$  is reset to 0 at the start of each strip (for line strips), and before every line segment in a group of independent segments.

If the line segment has been clipped, then the value of  $s$  at the beginning of the line segment is implementation-dependent.

The line stipple factor and pattern are specified by the `VkPipelineRasterizationLineStateCreateInfoEXT::lineStippleFactor` and `VkPipelineRasterizationLineStateCreateInfoEXT::lineStipplePattern` members of the currently active pipeline, if the pipeline was not created with `VK_DYNAMIC_STATE_LINE_STIPPLE_EXT` enabled.

Otherwise, the line stipple factor and pattern are set by calling `vkCmdSetLineStippleEXT`:

```
void vkCmdSetLineStippleEXT(
    VkCommandBuffer                                commandBuffer,
    uint32_t                                         lineStippleFactor,
    uint16_t                                         lineStipplePattern);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `lineStippleFactor` is the repeat factor used in stippled line rasterization.
- `lineStipplePattern` is the bit pattern used in stippled line rasterization.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_LINE_STIPPLE_EXT` dynamic state enabled
- `lineStippleFactor` **must** be in the range [1,256]

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

### 26.10.4. Smooth Lines

If the `lineRasterizationMode` member of `VkPipelineRasterizationLineStateCreateInfoEXT` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, then lines are considered to be rectangles using the same geometry as for `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` lines. The rules for determining which pixels are covered are implementation-dependent, and **may** include nearby pixels where no sample locations are covered or where the rectangle doesn't intersect the pixel at all. For each pixel that is considered covered, the fragment computes a coverage value that approximates the area of the intersection of the rectangle with the pixel square, and this coverage value is multiplied into the color location 0's alpha value after fragment shading, as described in [Multisample Coverage](#).

### Note



The details of the rasterization rules and area calculation are left intentionally vague, to allow implementations to generate coverage and values that are aesthetically pleasing.

## 26.11. Polygons

A polygon results from the decomposition of a triangle strip, triangle fan or a series of independent triangles. Like points and line segments, polygon rasterization is controlled by several variables in the [VkPipelineRasterizationStateCreateInfo](#) structure.

### 26.11.1. Basic Polygon Rasterization

The first step of polygon rasterization is to determine whether the triangle is *back-facing* or *front-facing*. This determination is made based on the sign of the (clipped or unclipped) polygon's area computed in framebuffer coordinates. One way to compute this area is:

$$a = -\frac{1}{2} \sum_{i=0}^{n-1} x_f^i y_f^{i \oplus 1} - x_f^{i \oplus 1} y_f^i$$

where  $x_f^i$  and  $y_f^i$  are the x and y framebuffer coordinates of the  $i$ th vertex of the  $n$ -vertex polygon (vertices are numbered starting at zero for the purposes of this computation) and  $i \oplus 1$  is  $(i + 1) \bmod n$ .

The interpretation of the sign of  $a$  is determined by the [VkPipelineRasterizationStateCreateInfo::frontFace](#) property of the currently active pipeline. Possible values are:

```
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
    VK_FRONT_FACE_MAX_ENUM = 0x7FFFFFFF
} VkFrontFace;
```

- `VK_FRONT_FACE_COUNTER_CLOCKWISE` specifies that a triangle with positive area is considered front-facing.
- `VK_FRONT_FACE_CLOCKWISE` specifies that a triangle with negative area is considered front-facing.

Any triangle which is not front-facing is back-facing, including zero-area triangles.

Once the orientation of triangles is determined, they are culled according to the [VkPipelineRasterizationStateCreateInfo::cullMode](#) property of the currently active pipeline. Possible values are:

```

typedef enum VkCullModeFlagBits {
    VK_CULL_MODE_NONE = 0,
    VK_CULL_MODE_FRONT_BIT = 0x00000001,
    VK_CULL_MODE_BACK_BIT = 0x00000002,
    VK_CULL_MODE_FRONT_AND_BACK = 0x00000003,
    VK_CULL_MODE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkCullModeFlagBits;

```

- **VK\_CULL\_MODE\_NONE** specifies that no triangles are discarded
- **VK\_CULL\_MODE\_FRONT\_BIT** specifies that front-facing triangles are discarded
- **VK\_CULL\_MODE\_BACK\_BIT** specifies that back-facing triangles are discarded
- **VK\_CULL\_MODE\_FRONT\_AND\_BACK** specifies that all triangles are discarded.

Following culling, fragments are produced for any triangles which have not been discarded.

```

typedef VkFlags VkCullModeFlags;

```

**VkCullModeFlags** is a bitmask type for setting a mask of zero or more **VkCullModeFlagBits**.

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y framebuffer coordinates of the polygon's vertices is formed. Fragments are produced for any fragment area groups of pixels for which any sample points lie inside of this polygon. Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Special treatment is given to a sample whose sample location lies on a polygon edge. In such a case, if two polygons lie on either side of a common edge (with identical endpoints) on which a sample point lies, then exactly one of the polygons **must** result in a covered sample for that fragment during rasterization. As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, a, b, and c, each in the range [0,1], with  $a + b + c = 1$ . These coordinates uniquely specify any point p within the triangle or on the triangle's boundary as

$$p = a p_a + b p_b + c p_c$$

where  $p_a$ ,  $p_b$ , and  $p_c$  are the vertices of the triangle. a, b, and c are determined by:

$$a = \frac{A(pp_bp_c)}{A(p_ap_bp_c)}, \quad b = \frac{A(pp_ap_c)}{A(p_ap_bp_c)}, \quad c = \frac{A(pp_ap_b)}{A(p_ap_bp_c)},$$

where  $A(lmn)$  denotes the area in framebuffer coordinates of the triangle with vertices l, m, and n.

Denote an associated datum at  $p_a$ ,  $p_b$ , or  $p_c$  as  $f_a$ ,  $f_b$ , or  $f_c$ , respectively.

The value of an associated datum f for a fragment produced by rasterizing a triangle, whether it be a shader output or the clip w coordinate, **must** be determined using perspective interpolation:

$$f = \frac{af_a / w_a + bf_b / w_b + cf_c / w_c}{a / w_a + b / w_b + c / w_c}$$

where  $w_a$ ,  $w_b$ , and  $w_c$  are the clip w coordinates of  $p_a$ ,  $p_b$ , and  $p_c$ , respectively.  $a$ ,  $b$ , and  $c$  are the barycentric coordinates of the location at which the data are produced - this **must** be the location of the fragment center or the location of a sample. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment center **must** be used.

Depth values for triangles **must** be determined using linear interpolation:

$$z = a z_a + b z_b + c z_c$$

where  $z_a$ ,  $z_b$ , and  $z_c$  are the depth values of  $p_a$ ,  $p_b$ , and  $p_c$ , respectively.

The `NoPerspective` and `Flat` interpolation decorations **can** be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, `perspective interpolation` is performed as described above. When the `NoPerspective` decoration is used, `linear interpolation` is performed in the same fashion as for depth values, as described above. When the `Flat` decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of the `provoking vertex` corresponding to that primitive.

When the `VK_AMD_shader_explicit_vertex_parameter` device extension is enabled the `CustomInterpAMD` interpolation decoration **can** also be used with fragment shader inputs which indicate that the decorated inputs **can** only be accessed by the extended instruction `InterpolateAtVertexAMD` and allows accessing the value of the inputs for individual vertices of the primitive.

When the `fragmentShaderBarycentric` feature is enabled, the `PerVertexNV` interpolation decoration **can** also be used with fragment shader inputs which indicate that the decorated inputs are not interpolated and **can** only be accessed using an extra array dimension, where the extra index identifies one of the vertices of the primitive that produced the fragment.

For a polygon with more than three edges, such as are produced by clipping a triangle, a convex combination of the values of the datum at the polygon's vertices **must** be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it **must** be the case that at every fragment

$$f = \sum_{i=1}^n a_i f_i$$

where  $n$  is the number of vertices in the polygon and  $f_i$  is the value of  $f$  at vertex  $i$ . For each  $i$ ,  $0 \leq a_i \leq 1$  and  $\sum_{i=1}^n a_i = 1$ . The values of  $a_i$  **may** differ from fragment to fragment, but at vertex  $i$ ,  $a_i = 1$  and  $a_j = 0$  for  $j \neq i$ .

### Note

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation [triangle\_perspective\_interpolation] are iterated independently and a division performed for each fragment).



## 26.11.2. Polygon Mode

Possible values of the `VkPipelineRasterizationStateCreateInfo::polygonMode` property of the currently active pipeline, specifying the method of rasterization for polygons, are:

```
typedef enum VkPolygonMode {
    VK_POLYGON_MODE_FILL = 0,
    VK_POLYGON_MODE_LINE = 1,
    VK_POLYGON_MODE_POINT = 2,
    VK_POLYGON_MODE_FILL_RECTANGLE_NV = 1000153000,
    VK_POLYGON_MODE_MAX_ENUM = 0x7FFFFFFF
} VkPolygonMode;
```

- `VK_POLYGON_MODE_POINT` specifies that polygon vertices are drawn as points.
- `VK_POLYGON_MODE_LINE` specifies that polygon edges are drawn as line segments.
- `VK_POLYGON_MODE_FILL` specifies that polygons are rendered using the polygon rasterization rules in this section.
- `VK_POLYGON_MODE_FILL_RECTANGLE_NV` specifies that polygons are rendered using polygon rasterization rules, modified to consider a sample within the primitive if the sample location is inside the axis-aligned bounding box of the triangle after projection. Note that the barycentric weights used in attribute interpolation **can** extend outside the range [0,1] when these primitives are shaded. Special treatment is given to a sample position on the boundary edge of the bounding box. In such a case, if two rectangles lie on either side of a common edge (with identical endpoints) on which a sample position lies, then exactly one of the triangles **must** produce a fragment that covers that sample during rasterization.

Polygons rendered in `VK_POLYGON_MODE_FILL_RECTANGLE_NV` mode **may** be clipped by the frustum or by user clip planes. If clipping is applied, the triangle is culled rather than clipped.

Area calculation and facingness are determined for `VK_POLYGON_MODE_FILL_RECTANGLE_NV` mode using the triangle's vertices.

These modes affect only the final rasterization of polygons: in particular, a polygon's vertices are shaded and the polygon is clipped and possibly culled before these modes are applied.

### 26.11.3. Depth Bias

The depth values of all fragments generated by the rasterization of a polygon **can** be offset by a single value that is computed for that polygon. This behavior is controlled by the `depthBiasEnable`, `depthBiasConstantFactor`, `depthBiasClamp`, and `depthBiasSlopeFactor` members of `VkPipelineRasterizationStateCreateInfo`, or by the corresponding parameters to the `vkCmdSetDepthBias` command if depth bias state is dynamic.

```
void vkCmdSetDepthBias(  
    VkCommandBuffer  
    float  
    float  
    float  
        commandBuffer,  
        depthBiasConstantFactor,  
        depthBiasClamp,  
        depthBiasSlopeFactor);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthBiasConstantFactor` is a scalar factor controlling the constant depth value added to each fragment.
- `depthBiasClamp` is the maximum (or minimum) depth bias of a fragment.
- `depthBiasSlopeFactor` is a scalar factor applied to a fragment's slope in depth bias calculations.

If `depthBiasEnable` is `VK_FALSE`, no depth bias is applied and the fragment's depth values are unchanged.

`depthBiasSlopeFactor` scales the maximum depth slope of the polygon, and `depthBiasConstantFactor` scales an implementation-dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the depth bias value which is then clamped to a minimum or maximum value specified by `depthBiasClamp`. `depthBiasSlopeFactor`, `depthBiasConstantFactor`, and `depthBiasClamp` **can** each be positive, negative, or zero.

The maximum depth slope  $m$  of a triangle is

$$m = \sqrt{\left(\frac{\partial z_f}{\partial x_f}\right)^2 + \left(\frac{\partial z_f}{\partial y_f}\right)^2}$$

where  $(x_f, y_f, z_f)$  is a point on the triangle.  $m$  **may** be approximated as

$$m = \max\left(\left|\frac{\partial z_f}{\partial x_f}\right|, \left|\frac{\partial z_f}{\partial y_f}\right|\right).$$

The minimum resolvable difference  $r$  is an implementation-dependent parameter that depends on the depth buffer representation. It is the smallest difference in framebuffer coordinate  $z$  values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but  $z_f$  values that differ by  $r$ , will have distinct depth values.

For fixed-point depth buffer representations,  $r$  is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum

exponent, e, in the range of z values spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r, for the given primitive is defined as

$$r = 2^{e-n}$$

If a triangle is rasterized using the `VK_POLYGON_MODE_FILL_RECTANGLE_NV` polygon mode, then this minimum resolvable difference **may** not be resolvable for samples outside of the triangle, where the depth is extrapolated.

If no depth buffer is present, r is undefined.

The bias value o for a polygon is

$$o = \text{dbclamp}(m \times \text{depthBiasSlopeFactor} + r \times \text{depthBiasConstantFactor})$$

where  $\text{dbclamp}(x) = \begin{cases} x & \text{depthBiasClamp} = 0 \text{ or } \text{NaN} \\ \min(x, \text{depthBiasClamp}) & \text{depthBiasClamp} > 0 \\ \max(x, \text{depthBiasClamp}) & \text{depthBiasClamp} < 0 \end{cases}$

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range [0,1], and o is applied to depth values in the same range.

For fixed-point depth buffers, fragment depth values are always limited to the range [0,1] by clamping after depth bias addition is performed. Unless the `VK_EXT_depth_range_unrestricted` extension is enabled, fragment depth values are clamped even when the depth buffer uses a floating-point representation.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_DEPTH_BIAS` dynamic state enabled
- If the `depth bias clamping` feature is not enabled, `depthBiasClamp` **must** be `0.0`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

### 26.11.4. Conservative Rasterization

Polygon rasterization **can** be made conservative by setting `conservativeRasterizationMode` to `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` or `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` in `VkPipelineRasterizationConservativeStateCreateInfoEXT`. The `VkPipelineRasterizationConservativeStateCreateInfoEXT` state is set by adding an instance of this structure to the `pNext` chain of an instance of the `VkPipelineRasterizationStateCreateInfo` structure when creating the graphics pipeline. Enabling these modes also affects line and point rasterization if the implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativePointAndLineRasterization` to `VK_TRUE`.

`VkPipelineRasterizationConservativeStateCreateInfoEXT` is defined as:

```
typedef struct VkPipelineRasterizationConservativeStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationConservativeStateCreateInfoFlagsEXT flags;
    VkConservativeRasterizationModeEXT conservativeRasterizationMode;
    float extraPrimitiveOverestimationSize;
} VkPipelineRasterizationConservativeStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `conservativeRasterizationMode` is the conservative rasterization mode to use.
- `extraPrimitiveOverestimationSize` is the extra size in pixels to increase the generating primitive during conservative rasterization at each of its edges in `X` and `Y` equally in screen space beyond the base overestimation specified in `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::primitiveOverestimationSize`.

## Valid Usage

- `extraPrimitiveOverestimationSize` **must** be in the range of `0.0` to `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::maxExtraPrimitiveOverestimationSize` inclusive

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT`
- `flags` **must** be `0`
- `conservativeRasterizationMode` **must** be a valid `VkConservativeRasterizationModeEXT` value

```
typedef VkFlags VkPipelineRasterizationConservativeStateCreateInfoEXT;
```

`VkPipelineRasterizationConservativeStateCreateInfoEXT` is a bitmask type for setting a mask, but is currently reserved for future use.

Possible values of `VkPipelineRasterizationConservativeStateCreateInfoEXT`::`conservativeRasterizationMode`, specifying the conservative rasterization mode are:

```
typedef enum VkConservativeRasterizationModeEXT {
    VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT = 0,
    VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT = 1,
    VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT = 2,
    VK_CONSERVATIVE_RASTERIZATION_MODE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkConservativeRasterizationModeEXT;
```

- `VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT` specifies that conservative rasterization is disabled and rasterization proceeds as normal.
- `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` specifies that conservative rasterization is enabled in overestimation mode.
- `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` specifies that conservative rasterization is enabled in underestimation mode.

When overestimate conservative rasterization is enabled, rather than evaluating coverage at individual sample locations, a determination is made of whether any portion of the pixel (including its edges and corners) is covered by the primitive. If any portion of the pixel is covered, then all bits of the coverage sample mask for the fragment corresponding to that pixel are enabled. If the render pass has a fragment density map attachment and any bit of the coverage sample mask for the fragment is enabled, then all bits of the coverage sample mask for the fragment are enabled.

If the implementation supports `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`  
`::conservativeRasterizationPostDepthCoverage` and the `PostDepthCoverage` execution mode is  
specified the `SampleMask` built-in input variable will reflect the coverage after the early per-fragment  
depth and stencil tests are applied.

For the purposes of evaluating which pixels are covered by the primitive, implementations **can**  
increase the size of the primitive by up to `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`  
`::primitiveOverestimationSize` pixels at each of the primitive edges. This **may** increase the number  
of fragments generated by this primitive and represents an overestimation of the pixel coverage.

This overestimation size can be increased further by setting the `extraPrimitiveOverestimationSize`  
value above `0.0` in steps of `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`  
`::extraPrimitiveOverestimationSizeGranularity` up to and including  
`VkPhysicalDeviceConservativeRasterizationPropertiesEXT::extraPrimitiveOverestimationSize`. This  
will further increase the number of fragments generated by this primitive.

The actual precision of the overestimation size used for conservative rasterization **may** vary  
between implementations and produce results that only approximate the  
`primitiveOverestimationSize` and `extraPrimitiveOverestimationSizeGranularity` properties.  
Implementations **may** especially vary these approximations when the render pass has a fragment  
density map and the fragment area covers multiple pixels.

For triangles if `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` is enabled, fragments will be  
generated if the primitive area covers any portion of any pixel inside the fragment area, including  
their edges or corners. The tie-breaking rule described in [Basic Polygon Rasterization](#) does not  
apply during conservative rasterization and coverage is set for all fragments generated from shared  
edges of polygons. Degenerate triangles that evaluate to zero area after rasterization, even for  
pixels containing a vertex or edge of the zero-area polygon, will be culled if  
`VkPhysicalDeviceConservativeRasterizationPropertiesEXT::degenerateTrianglesRasterized` is `VK_FALSE`  
or will generate fragments if `degenerateTrianglesRasterized` is `VK_TRUE`. The fragment input values  
for these degenerate triangles take their attribute and depth values from the provoking vertex.  
Degenerate triangles are considered backfacing and the application **can** enable backface culling if  
desired. Triangles that are zero area before rasterization **may** be culled regardless.

For lines if `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` is enabled, and the  
implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`  
`::conservativePointAndLineRasterization` to `VK_TRUE`, fragments will be generated if the line covers  
any portion of any pixel inside the fragment area, including their edges or corners. Degenerate  
lines that evaluate to zero length after rasterization will be culled if  
`VkPhysicalDeviceConservativeRasterizationPropertiesEXT::degenerateLinesRasterized` is `VK_FALSE`  
or will generate fragments if `degenerateLinesRasterized` is `VK_TRUE`. The fragments input values  
for these degenerate lines take their attribute and depth values from the provoking vertex. Lines that  
are zero length before rasterization **may** be culled regardless.

For points if `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` is enabled, and the  
implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`  
`::conservativePointAndLineRasterization` to `VK_TRUE`, fragments will be generated if the point square  
covers any portion of any pixel inside the fragment area, including their edges or corners.

When underestimate conservative rasterization is enabled, rather than evaluating coverage at individual sample locations, a determination is made of whether all of the pixel (including its edges and corners) is covered by the primitive. If the entire pixel is covered, then a fragment is generated with all bits of its coverage sample mask corresponding to the pixel enabled, otherwise the pixel is not considered covered even if some portion of the pixel is covered. The fragment is discarded if no pixels inside the fragment area are considered covered. If the render pass has a fragment density map attachment and any pixel inside the fragment area is not considered covered, then the fragment is discarded even if some pixels are considered covered.

If the implementation supports `VkPhysicalDeviceConservativeRasterizationPropertiesEXT` `::conservativeRasterizationPostDepthCoverage` and the `PostDepthCoverage` execution mode is specified the `SampleMask` built-in input variable will reflect the coverage after the early per-fragment depth and stencil tests are applied.

For triangles, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if any pixel inside the fragment area is fully covered by the generating primitive, including its edges and corners.

For lines, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will be generated if any pixel inside the fragment area, including its edges and corners, are entirely covered by the line.

For points, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if the point square covers the entirety of any pixel square inside the fragment area, including its edges or corners.

If the render pass has a fragment density map and `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if the entirety of all pixels inside the fragment area are covered by the generating primitive, line, or point.

For both overestimate and underestimate conservative rasterization modes a fragment has all of its pixel squares fully covered by the generating primitive **must** set `FullyCoveredEXT` to `VK_TRUE` if the implementation enables the `VkPhysicalDeviceConservativeRasterizationPropertiesEXT` `::fullyCoveredFragmentShaderInputVariable` feature.

When the use of a `shading rate image` results in fragments covering multiple pixels, coverage for conservative rasterization is still evaluated on a per-pixel basis and may result in fragments with partial coverage. For fragment shader inputs decorated with `FullyCoveredEXT`, a fragment is considered fully covered if and only if all pixels in the fragment are fully covered by the generating primitive.

# Chapter 27. Fragment Operations

Fragment operations execute on a per-fragment or per-sample basis, affecting whether or how a fragment or sample is written to the framebuffer. Some operations execute [before fragment shading](#), and others [after](#). Fragment operations always adhere to [rasterization order](#).

## 27.1. Early Per-Fragment Tests

Once fragments are produced by rasterization, a number of per-fragment operations are performed prior to fragment shader execution. If a fragment is discarded during any of these operations, it will not be processed by any subsequent stage, including fragment shader execution.

The [scissor test](#), [exclusive scissor test](#), and [sample mask generation](#) are always performed during early fragment tests.

Fragment operations are performed in the following order:

- the discard rectangles test (see [Discard Rectangles Test](#))
- the scissor test (see [Scissor Test](#))
- the exclusive scissor test (see [Exclusive Scissor Test](#))
- multisample fragment operations (see [Sample Mask](#))

If early per-fragment operations are [enabled by the fragment shader](#), these operations are also performed:

- [Depth bounds test](#)
- [Stencil test](#)
- [Depth test](#)
- [Representative fragment test](#)
- [Sample counting](#) for occlusion queries

If post-depth coverage operation is [enabled by the fragment shader](#), the [SampleMask](#) coverage is determined after the early stencil and depth tests.

## 27.2. Discard Rectangles Test

The discard rectangles test determines if fragment's framebuffer coordinates ( $x_f, y_f$ ) are inclusive or exclusive to a set of discard-space rectangles. The discard rectangles are set with the [VkPipelineDiscardRectangleStateCreateInfoEXT](#) pipeline state, which is defined as:

```

typedef struct VkPipelineDiscardRectangleStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineDiscardRectangleStateCreateFlagsEXT flags;
    VkDiscardRectangleModeEXT discardRectangleMode;
    uint32_t discardRectangleCount;
    const VkRect2D* pDiscardRectangles;
} VkPipelineDiscardRectangleStateCreateInfoEXT;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **discardRectangleMode** is the mode used to determine whether fragments that lie within the discard rectangle are discarded or not.
- **discardRectangleCount** is the number of discard rectangles used by the pipeline.
- **pDiscardRectangles** is a pointer to an array of **VkRect2D** structures, defining the discard rectangles. If the discard rectangle state is dynamic, this member is ignored.

## Valid Usage

- **discardRectangleCount** must be between **0** and **VkPhysicalDeviceDiscardRectanglePropertiesEXT::maxDiscardRectangles**, inclusive

## Valid Usage (Implicit)

- **sType** must be **VK\_STRUCTURE\_TYPE\_PIPELINE\_DISCARD\_RECTANGLE\_STATE\_CREATE\_INFO\_EXT**
- **flags** must be **0**
- **discardRectangleMode** must be a valid **VkDiscardRectangleModeEXT** value

```
typedef VkFlags VkPipelineDiscardRectangleStateCreateFlagsEXT;
```

**VkPipelineDiscardRectangleStateCreateFlagsEXT** is a bitmask type for setting a mask, but is currently reserved for future use.

The **VkPipelineDiscardRectangleStateCreateInfoEXT** state is set by adding an instance of this structure to the **pNext** chain of an instance of the **VkGraphicsPipelineCreateInfo** structure and setting the graphics pipeline state with **vkCreateGraphicsPipelines**.

If the bound pipeline state object was not created with the **VK\_DYNAMIC\_STATE\_DISCARD\_RECTANGLE\_EXT** dynamic state enabled, discard rectangles are specified using the **pDiscardRectangles** member of **VkPipelineDiscardRectangleStateCreateInfoEXT** linked to the pipeline state object.

If the pipeline state object was created with the `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT` dynamic state enabled, the discard rectangles are dynamically set and changed with the command:

```
void vkCmdSetDiscardRectangleEXT(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkRect2D*  
                                commandBuffer,  
                                firstDiscardRectangle,  
                                discardRectangleCount,  
                                pDiscardRectangles);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstDiscardRectangle` is the index of the first discard rectangle whose state is updated by the command.
- `discardRectangleCount` is the number of discard rectangles whose state are updated by the command.
- `pDiscardRectangles` is a pointer to an array of `VkRect2D` structures specifying discard rectangles.

The discard rectangle taken from element `i` of `pDiscardRectangles` replace the current state for the discard rectangle index `firstDiscardRectangle + i`, for `i` in `[0, discardRectangleCount]`.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT` dynamic state enabled
- The sum of `firstDiscardRectangle` and `discardRectangleCount` **must** be less than or equal to `VkPhysicalDeviceDiscardRectanglePropertiesEXT::maxDiscardRectangles`
- The `x` and `y` member of `offset` in each `VkRect2D` element of `pDiscardRectangles` **must** be greater than or equal to `0`
- Evaluation of `(offset.x + extent.width)` in each `VkRect2D` element of `pDiscardRectangles` **must** not cause a signed integer addition overflow
- Evaluation of `(offset.y + extent.height)` in each `VkRect2D` element of `pDiscardRectangles` **must** not cause a signed integer addition overflow

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pDiscardRectangles` **must** be a valid pointer to an array of `discardRectangleCount` `VkRect2D` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `discardRectangleCount` **must** be greater than `0`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

The `VkOffset2D::x` and `VkOffset2D::y` values of the discard rectangle `VkRect2D` specify the upper-left origin of the discard rectangle box. The lower-right corner of the discard rectangle box is specified as the `VkExtent2D::width` and `VkExtent2D::height` from the upper-left origin.

If `offset.x ≤ xf < offset.x + extent.width` and `offset.y ≤ yf < offset.y + extent.height` for the selected discard rectangle, then the fragment is within the discard rectangle box. When the discard rectangle mode is `VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT` a fragment within at least one of the active discard rectangle boxes passes the discard rectangle test; otherwise the fragment fails the discard rectangle test and is discarded. When the discard rectangle mode is `VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT` a fragment within at least one of the active discard rectangle boxes fails the discard rectangle test, and the fragment is discarded; otherwise the fragment passes the discard rectangles test. The discard rectangles test only applies to [drawing commands](#), not to other commands like clears or copies.

Possible values of `VkPipelineDiscardRectangleStateCreateInfoEXT::discardRectangleMode`, specifying the behavior of the discard rectangle test, are:

```
typedef enum VkDiscardRectangleModeEXT {
    VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT = 0,
    VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT = 1,
    VK_DISCARD_RECTANGLE_MODE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDiscardRectangleModeEXT;
```

- `VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT` specifies that a fragment within any discard rectangle satisfies the test.
- `VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT` specifies that a fragment not within any of the discard rectangles satisfies the test.

When the use of a shading rate image results in a fragment covering multiple pixels, the discard rectangle test is performed independently for each pixel in the fragment. If a pixel covered by a fragment fails the discard rectangle test, all samples in the fragment associated with that pixel are

treated as not covered. If the discard rectangle test results in a fragment with no samples covered, that fragment is discarded.

## 27.3. Scissor Test

The scissor test determines if a fragment's framebuffer coordinates ( $x, y$ ) lie within the scissor rectangle corresponding to the viewport index (see [Controlling the Viewport](#)) used by the primitive that generated the fragment. If the pipeline state object is created without `VK_DYNAMIC_STATE_SCISSOR` enabled then the scissor rectangles are set by the `VkPipelineViewportStateCreateInfo` state of the pipeline state object. Otherwise, to dynamically set the scissor rectangles call:

```
void vkCmdSetScissor(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkRect2D*  
                                commandBuffer,  
                                firstScissor,  
                                scissorCount,  
                                pScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstScissor` is the index of the first scissor whose state is updated by the command.
- `scissorCount` is the number of scissors whose rectangles are updated by the command.
- `pScissors` is a pointer to an array of `VkRect2D` structures defining scissor rectangles.

The scissor rectangles taken from element  $i$  of `pScissors` replace the current state for the scissor index `firstScissor + i`, for  $i$  in  $[0, \text{scissorCount}]$ .

Each scissor rectangle is described by a `VkRect2D` structure, with the `offset.x` and `offset.y` values determining the upper left corner of the scissor rectangle, and the `extent.width` and `extent.height` values determining the size in pixels.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_SCISSOR` dynamic state enabled
- `firstScissor` **must** be less than `VkPhysicalDeviceLimits::maxViewports`
- The sum of `firstScissor` and `scissorCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- If the `multiple viewports` feature is not enabled, `firstScissor` **must** be 0
- If the `multiple viewports` feature is not enabled, `scissorCount` **must** be 1
- The `x` and `y` members of `offset` member of any element of `pScissors` **must** be greater than or equal to 0
- Evaluation of `(offset.x + extent.width)` **must** not cause a signed integer addition overflow for any element of `pScissors`
- Evaluation of `(offset.y + extent.height)` **must** not cause a signed integer addition overflow for any element of `pScissors`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pScissors` **must** be a valid pointer to an array of `scissorCount` `VkRect2D` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `scissorCount` **must** be greater than 0

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

If `offset.x ≤ xf < offset.x + extent.width` and `offset.y ≤ yf < offset.y + extent.height` for the selected scissor rectangle, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. For points, lines, and polygons, the scissor rectangle for a primitive is selected in the same manner as the viewport (see [Controlling the Viewport](#)). The scissor rectangles test only applies to [drawing commands](#), not to other commands like clears or copies.

It is legal for `offset.x + extent.width` or `offset.y + extent.height` to exceed the dimensions of the framebuffer - the scissor test still applies as defined above. Rasterization does not produce fragments outside of the framebuffer, so such fragments never have the scissor test performed on them.

The scissor test is always performed. Applications **can** effectively disable the scissor test by specifying a scissor rectangle that encompasses the entire framebuffer.

When the use of a shading rate image results in a fragment covering multiple pixels, the scissor test is performed independently for each pixel in the fragment. If a pixel covered by a fragment fails the scissor test, all samples in the fragment associated with that pixel are treated as not covered. If the scissor test results in a fragment with no samples covered, that fragment is discarded.

## 27.4. Exclusive Scissor Test

The exclusive scissor test determines if a pixel's framebuffer coordinates ( $x_f, y_f$ ) lie outside the exclusive scissor rectangle corresponding to the viewport index (see [Controlling the Viewport](#)) used by the primitive that generated the fragment. The exclusive scissor test behaves identically to the [scissor test](#), except that it passes only if the pixel is outside the rectangle instead of passing if the pixel is inside the rectangle.

If the `pNext` chain of `VkPipelineViewportStateCreateInfo` includes a `VkPipelineViewportExclusiveScissorStateCreateInfoNV` structure, then that structure includes parameters that affect the exclusive scissor test.

The `VkPipelineViewportExclusiveScissorStateCreateInfoNV` structure is defined as:

```
typedef struct VkPipelineViewportExclusiveScissorStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    uint32_t exclusiveScissorCount;
    const VkRect2D* pExclusiveScissors;
} VkPipelineViewportExclusiveScissorStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `exclusiveScissorCount` is the number of exclusive scissor rectangles used by the pipeline.
- `pExclusiveScissors` is a pointer to an array of `VkRect2D` structures defining exclusive scissor rectangles. If the exclusive scissor state is dynamic, this member is ignored.

If this structure is not present, `exclusiveScissorCount` is considered to be `0` and the exclusive scissor

test is disabled.

## Valid Usage

- If the [multiple viewports](#) feature is not enabled, `exclusiveScissorCount` **must** be `0` or `1`
- `exclusiveScissorCount` **must** be less than or equal to `VkPhysicalDeviceLimits::maxViewports`
- `exclusiveScissorCount` **must** be `0` or identical to the `viewportCount` member of `VkPipelineViewportStateCreateInfo`
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV` and `exclusiveScissorCount` is not `0`, `pExclusiveScissors` **must** be a valid pointer to an array of `exclusiveScissorCount` `VkRect2D` structures

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_EXCLUSIVE_SCISSOR_STATE_CREATE_INFO_NV`
- If `exclusiveScissorCount` is not `0`, and `pExclusiveScissors` is not `NULL`, `pExclusiveScissors` **must** be a valid pointer to an array of `exclusiveScissorCount` `VkRect2D` structures

If the pipeline state object is created with `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV` enabled, then the exclusive scissor rectangles are set by:

```
void vkCmdSetExclusiveScissorNV(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    const VkRect2D*  
                                commandBuffer,  
                                firstExclusiveScissor,  
                                exclusiveScissorCount,  
                                pExclusiveScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstExclusiveScissor` is the index of the first exclusive scissor rectangle whose state is updated by the command.
- `exclusiveScissorCount` is the number of exclusive scissor rectangles updated by the command.
- `pExclusiveScissors` is a pointer to an array of `VkRect2D` structures defining exclusive scissor rectangles.

The scissor rectangles taken from element `i` of `pExclusiveScissors` replace the current state for the scissor index `firstExclusiveScissor + i`, for `i` in `[0, exclusiveScissorCount]`.

Each scissor rectangle is described by a `VkRect2D` structure, with the `offset.x` and `offset.y` values determining the upper left corner of the scissor rectangle, and the `extent.width` and `extent.height` values determining the size in pixels.

## Valid Usage

- The `exclusive scissor` feature **must** be enabled.
- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV` dynamic state enabled
- `firstExclusiveScissor` **must** be less than `VkPhysicalDeviceLimits::maxViewports`
- The sum of `firstExclusiveScissor` and `exclusiveScissorCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- If the `multiple viewports` feature is not enabled, `firstExclusiveScissor` **must** be 0
- If the `multiple viewports` feature is not enabled, `exclusiveScissorCount` **must** be 1
- The `x` and `y` members of `offset` in each member of `pExclusiveScissors` **must** be greater than or equal to 0
- Evaluation of (`offset.x + extent.width`) for each member of `pExclusiveScissors` **must** not cause a signed integer addition overflow
- Evaluation of (`offset.y + extent.height`) for each member of `pExclusiveScissors` **must** not cause a signed integer addition overflow

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pExclusiveScissors` **must** be a valid pointer to an array of `exclusiveScissorCount` `VkRect2D` structures
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `exclusiveScissorCount` **must** be greater than 0

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

If `offset.x ≤ xf < offset.x + extent.width` and `offset.y ≤ yf < offset.y + extent.height` for the selected exclusive scissor rectangle, then the exclusive scissor test fails and the fragment is discarded. Otherwise, the exclusive scissor test passes. For points, lines, and polygons, the exclusive scissor rectangle for a primitive is selected in the same manner as the viewport (see [Controlling the Viewport](#)). The exclusive scissor test only applies to [drawing commands](#), not to other commands like clears or copies.

It is legal for `offset.x + extent.width` or `offset.y + extent.height` to exceed the dimensions of the framebuffer - the exclusive scissor test still applies as defined above. Rasterization does not produce fragments outside of the framebuffer, so such fragments never have the exclusive scissor test performed on them.

The exclusive scissor test is performed if and only if the current pipeline was created with a non-zero `exclusiveScissorCount`. Applications [can](#) effectively disable the exclusive scissor test for specific viewports by specifying a scissor rectangle with a width or height of zero.

When the use of a shading rate image results in a fragment covering multiple pixels, the exclusive scissor test is performed independently for each pixel in the fragment. If a pixel covered by a fragment fails the exclusive scissor test, all samples in the fragment associated with that pixel are treated as not covered. If the exclusive scissor test results in a fragment with no samples covered, that fragment is discarded.

## 27.5. Sample Mask

This step modifies fragment coverage values based on the values in the `pSampleMask` array member of `VkPipelineMultisampleStateCreateInfo`, as described previously in section [Graphics Pipelines](#).

`pSampleMask` contains an array of static coverage information that is [ANDed](#) with the coverage information generated during rasterization. Bits that are zero disable coverage for the corresponding sample. Bit B of mask word M corresponds to sample  $32 \times M + B$ . The array is sized to a length of `rasterizationSamples / 32` words. If `pSampleMask` is `NULL`, it is treated as if the mask has all bits enabled, i.e. no coverage is removed from fragments.

The elements of the sample mask array are of type `VkSampleMask`, each representing 32 bits of coverage information:

```
typedef uint32_t VkSampleMask;
```

## 27.6. Early Fragment Test Mode

The depth bounds test, stencil test, depth test, representative fragment test, and occlusion query sample counting are performed before fragment shading if and only if early fragment tests are enabled by the fragment shader (see [Early Fragment Tests](#)). When early per-fragment operations are enabled, these operations are performed prior to fragment shader execution, and the stencil buffer, depth buffer, and occlusion query sample counts will be updated accordingly; these operations will not be performed again after fragment shader execution.

If a pipeline's fragment shader has early fragment tests disabled, these operations are performed only after fragment program execution, in the order described below. If a pipeline does not contain a fragment shader, these operations are performed only once.

If early fragment tests are enabled, any depth value computed by the fragment shader has no effect. Additionally, the depth test (including depth writes), stencil test (including stencil writes) and sample counting operations are performed even for fragments or samples that would be discarded after fragment shader execution due to per-fragment operations such as alpha-to-coverage tests, or due to the fragment being discarded by the shader itself.

## 27.7. Late Per-Fragment Tests

After programmable fragment processing, per-fragment operations are performed before blending and color output to the framebuffer.

A fragment is produced by rasterization with framebuffer coordinates of  $(x_f, y_f)$  and depth  $z$ , as described in [Rasterization](#). The fragment is then modified by programmable fragment processing, which adds associated data as described in [Shaders](#). The fragment is then further modified, and possibly discarded by the late per-fragment operations described in this chapter. Finally, if the fragment was not discarded, it is used to update the framebuffer at the fragment's framebuffer coordinates for any samples that remain covered.

The depth bounds test, stencil test, and depth test are performed for each sample, rather than just once for each fragment. Stencil and depth operations are performed for a sample only if that sample's fragment coverage bit is a value of 1 when the fragment executes the corresponding stage of the graphics pipeline. If the corresponding coverage bit is 0, no operations are performed for that sample. Failure of the depth bounds, stencil, or depth test results in termination of the processing of that sample by means of disabling coverage for that sample, rather than discarding of the fragment. If, at any point, a fragment's coverage becomes zero for all samples, then the fragment is discarded. All operations are performed on the depth and stencil values stored in the depth/stencil attachment of the framebuffer. The contents of the color attachments are not modified at this point.

The depth bounds test, stencil test, depth test, and occlusion query operations described in [Depth Bounds Test](#), [Stencil Test](#), [Depth Test](#), [Sample Counting](#) are instead performed prior to fragment processing, as described in [Early Fragment Test Mode](#), if requested by the fragment shader.

## 27.8. Mixed attachment samples

When the `VK_AMD_mixed_attachment_samples` extension is enabled, special rules apply to per-

fragment operations when the number of samples of the color attachments differs from the number of samples of the depth/stencil attachment used in a subpass.

Let C be the number of color attachment samples and D be the number of depth/stencil attachment samples used by a given subpass.

If  $C < D$  then only the first C number of samples are guaranteed to have a corresponding fragment shader invocation and thus a corresponding color output value, unless the fragment shaders produce inputs to the late per-fragment tests (e.g. by outputting to a variable decorated with the `FragDepth` built-in decoration). Implementations are allowed to produce fragment shader invocations for samples with indices greater than or equal to C but (other than potential side effects) the color outputs of fragment shader invocations corresponding to such samples are discarded.

## 27.9. Multisample Coverage

If a fragment shader is active and its entry point's interface includes a built-in output variable decorated with `SampleMask` and also decorated with `OverrideCoverageNV` the fragment coverage is replaced with the sample mask bits set in the shader. Otherwise if the built-in output variable decorated with `SampleMask` is not also decorated with `OverrideCoverageNV` then the fragment coverage is `ANDed` with the bits of the sample mask to generate a new fragment coverage value. If such a fragment shader did not assign a value to `SampleMask` due to flow of control, the value `ANDed` with the fragment coverage is undefined. If no fragment shader is active, or if the active fragment shader does not include `SampleMask` in its interface, the fragment coverage is not modified.

Next, the fragment alpha and coverage values are modified based on the line coverage factor if the `lineRasterizationMode` member of the `VkPipelineRasterizationStateCreateInfo` structure is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, and the `alphaToCoverageEnable` and `alphaToOneEnable` members of the `VkPipelineMultisampleStateCreateInfo` structure.

All alpha values in this section refer only to the alpha component of the fragment shader output that has a `Location` and `Index` decoration of zero (see the `Fragment Output Interface` section). If that shader output has an integer or unsigned integer type, then these operations are skipped.

If the `lineRasterizationMode` member of the `VkPipelineRasterizationStateCreateInfo` structure is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` and the fragment came from a line segment, then the alpha value is replaced by multiplying it by the coverage factor for the fragment computed during `smooth line rasterization`.

If `alphaToCoverageEnable` is enabled, a temporary coverage value with `rasterizationSamples` bits is generated where each bit is determined by the fragment's alpha value. The temporary coverage value is then `ANDed` with the fragment coverage value to generate a new fragment coverage value.

No specific algorithm is specified for converting the alpha value to a temporary coverage mask. It is intended that the number of 1's in this value be proportional to the alpha value (clamped to [0,1]), with all 1's corresponding to a value of 1.0 and all 0's corresponding to 0.0. The algorithm `may` be different at different framebuffer coordinates.

**Note**



Using different algorithms at different framebuffer coordinates **may** help to avoid artifacts caused by regular coverage sample locations.

Next, if `alphaToOneEnable` is enabled, each alpha value is replaced by the maximum representable alpha value for fixed-point color buffers, or by 1.0 for floating-point buffers. Otherwise, the alpha values are not changed.

## 27.10. Depth and Stencil Operations

Pipeline state controlling the `depth bounds tests`, `stencil test`, and `depth test` is specified through the members of the `VkPipelineDepthStencilStateCreateInfo` structure.

The `VkPipelineDepthStencilStateCreateInfo` structure is defined as:

```
typedef struct VkPipelineDepthStencilStateCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineDepthStencilStateCreateFlags flags;
    VkBool32                            depthTestEnable;
    VkBool32                            depthWriteEnable;
    VkCompareOp                         depthCompareOp;
    VkBool32                            depthBoundsTestEnable;
    VkBool32                            stencilTestEnable;
    VkStencilOpState                    front;
    VkStencilOpState                    back;
    float                               minDepthBounds;
    float                               maxDepthBounds;
} VkPipelineDepthStencilStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `depthTestEnable` controls whether `depth testing` is enabled.
- `depthWriteEnable` controls whether `depth writes` are enabled when `depthTestEnable` is `VK_TRUE`. Depth writes are always disabled when `depthTestEnable` is `VK_FALSE`.
- `depthCompareOp` is the comparison operator used in the `depth test`.
- `depthBoundsTestEnable` controls whether `depth bounds testing` is enabled.
- `stencilTestEnable` controls whether `stencil testing` is enabled.
- `front` and `back` control the parameters of the `stencil test`.
- `minDepthBounds` and `maxDepthBounds` define the range of values used in the `depth bounds test`.

## Valid Usage

- If the [depth bounds testing](#) feature is not enabled, `depthBoundsTestEnable` must be `VK_FALSE`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- `depthCompareOp` must be a valid [VkCompareOp](#) value
- `front` must be a valid [VkStencilOpState](#) structure
- `back` must be a valid [VkStencilOpState](#) structure

```
typedef VkFlags VkPipelineDepthStencilStateCreateInfo;
```

`VkPipelineDepthStencilStateCreateInfo` is a bitmask type for setting a mask, but is currently reserved for future use.

## 27.11. Depth Bounds Test

The depth bounds test conditionally disables coverage of a sample based on the outcome of a comparison between the value  $z_a$  in the depth attachment at location  $(x_b, y_b)$  (for the appropriate sample) and a range of values. The test is enabled or disabled by the `depthBoundsTestEnable` member of [VkPipelineDepthStencilStateCreateInfo](#): If the pipeline state object is created without the `VK_DYNAMIC_STATE_DEPTH_BOUNDS` dynamic state enabled then the range of values used in the depth bounds test are defined by the `minDepthBounds` and `maxDepthBounds` members of the [VkPipelineDepthStencilStateCreateInfo](#) structure. Otherwise, to dynamically set the depth bounds range values call:

```
void vkCmdSetDepthBounds(  
    VkCommandBuffer  
    float  
    float  
    commandBuffer,  
    minDepthBounds,  
    maxDepthBounds);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `minDepthBounds` is the lower bound of the range of depth values used in the depth bounds test.
- `maxDepthBounds` is the upper bound of the range.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_DEPTH_BOUNDS` dynamic state enabled
- Unless the `VK_EXT_depth_range_unrestricted` extension is enabled `minDepthBounds` **must** be between `0.0` and `1.0`, inclusive
- Unless the `VK_EXT_depth_range_unrestricted` extension is enabled `maxDepthBounds` **must** be between `0.0` and `1.0`, inclusive

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

If  $\text{minDepthBounds} \leq z_a \leq \text{maxDepthBounds}$ , then the depth bounds test passes. Otherwise, the test fails and the sample's coverage bit is cleared in the fragment. If there is no depth framebuffer attachment or if the depth bounds test is disabled, it is as if the depth bounds test always passes.

## 27.12. Stencil Test

The stencil test conditionally disables coverage of a sample based on the outcome of a comparison between the stencil value in the depth/stencil attachment at location  $(x_f, y_f)$  (for the appropriate sample) and a reference value. The stencil test also updates the value in the stencil attachment, depending on the test state, the stencil value and the stencil write masks. The test is enabled or disabled by the `stencilTestEnable` member of `VkPipelineDepthStencilStateCreateInfo`.

When disabled, the stencil test and associated modifications are not made, and the sample's coverage is not modified.

The stencil test is controlled by the `front` and `back` members of `VkPipelineDepthStencilStateCreateInfo`, which are of type `VkStencilOpState`.

The `VkStencilOpState` structure is defined as:

```
typedef struct VkStencilOpState {
    VkStencilOp    failOp;
    VkStencilOp    passOp;
    VkStencilOp    depthFailOp;
    VkCompareOp    compareOp;
    uint32_t       compareMask;
    uint32_t       writeMask;
    uint32_t       reference;
} VkStencilOpState;
```

- `failOp` is a `VkStencilOp` value specifying the action performed on samples that fail the stencil test.
- `passOp` is a `VkStencilOp` value specifying the action performed on samples that pass both the depth and stencil tests.
- `depthFailOp` is a `VkStencilOp` value specifying the action performed on samples that pass the stencil test and fail the depth test.
- `compareOp` is a `VkCompareOp` value specifying the comparison operator used in the stencil test.
- `compareMask` selects the bits of the unsigned integer stencil values participating in the stencil test.
- `writeMask` selects the bits of the unsigned integer stencil values updated by the stencil test in the stencil framebuffer attachment.
- `reference` is an integer reference value that is used in the unsigned stencil comparison.

## Valid Usage (Implicit)

- `failOp` **must** be a valid `VkStencilOp` value
- `passOp` **must** be a valid `VkStencilOp` value
- `depthFailOp` **must** be a valid `VkStencilOp` value
- `compareOp` **must** be a valid `VkCompareOp` value

There are two sets of stencil-related state, the front stencil state set and the back stencil state set. Stencil tests and writes use the front set of stencil state when processing front-facing fragments and use the back set of stencil state when processing back-facing fragments. Fragments rasterized from non-polygon primitives (points and lines) are always considered front-facing. Fragments rasterized from polygon primitives inherit their facingness from the polygon, even if the polygon is rasterized as points or lines due to the current `VkPolygonMode`. Whether a polygon is front- or back-facing is

determined in the same manner used for face culling (see [Basic Polygon Rasterization](#)).

The operation of the stencil test is also affected by the `compareMask`, `writeMask`, and `reference` members of `VkStencilOpState` set in the pipeline state object if the pipeline state object is created without the `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK`, `VK_DYNAMIC_STATE_STENCIL_WRITE_MASK`, and `VK_DYNAMIC_STATE_STENCIL_REFERENCE` dynamic states enabled, respectively.

If the pipeline state object is created with the `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK` dynamic state enabled, then to dynamically set the stencil compare mask call:

```
void vkCmdSetStencilCompareMask(  
    VkCommandBuffer  
    VkStencilFaceFlags  
    uint32_t  
                                commandBuffer,  
                                faceMask,  
                                compareMask);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the compare mask.
- `compareMask` is the new value to use as the stencil compare mask.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK` dynamic state enabled

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `faceMask` **must** be a valid combination of `VkStencilFaceFlagBits` values
- `faceMask` **must** not be `0`
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

Bits which **can** be set in the `vkCmdSetStencilCompareMask::faceMask` parameter, and similar parameters of other commands specifying which stencil state to update stencil masks for, are:

```
typedef enum VkStencilFaceFlagBits {
    VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
    VK_STENCIL_FACE_BACK_BIT = 0x00000002,
    VK_STENCIL_FACE_FRONT_AND_BACK = 0x00000003,
    VK_STENCIL_FRONT_AND_BACK = VK_STENCIL_FACE_FRONT_AND_BACK,
    VK_STENCIL_FACE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkStencilFaceFlagBits;
```

- `VK_STENCIL_FACE_FRONT_BIT` specifies that only the front set of stencil state is updated.
- `VK_STENCIL_FACE_BACK_BIT` specifies that only the back set of stencil state is updated.
- `VK_STENCIL_FACE_FRONT_AND_BACK` is the combination of `VK_STENCIL_FACE_FRONT_BIT` and `VK_STENCIL_FACE_BACK_BIT`, and specifies that both sets of stencil state are updated.

```
typedef VkFlags VkStencilFaceFlags;
```

`VkStencilFaceFlags` is a bitmask type for setting a mask of zero or more `VkStencilFaceFlagBits`.

If the pipeline state object is created with the `VK_DYNAMIC_STATE_STENCIL_WRITE_MASK` dynamic state enabled, then to dynamically set the stencil write mask call:

```
void vkCmdSetStencilWriteMask(
    VkCommandBuffer                                commandBuffer,
    VkStencilFaceFlags                            faceMask,
    uint32_t                                     writeMask);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the write mask, as described above for `vkCmdSetStencilCompareMask`.
- `writeMask` is the new value to use as the stencil write mask.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_STENCIL_WRITE_MASK` dynamic state enabled

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `faceMask` **must** be a valid combination of `VkStencilFaceFlagBits` values
- `faceMask` **must** not be `0`
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

If the pipeline state object is created with the `VK_DYNAMIC_STATE_STENCIL_REFERENCE` dynamic state enabled, then to dynamically set the stencil reference value call:

```
void vkCmdSetStencilReference(  
    VkCommandBuffer  
    VkStencilFaceFlags  
    uint32_t  
        commandBuffer,  
        faceMask,  
        reference);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the reference value, as described above for `vkCmdSetStencilCompareMask`.
- `reference` is the new value to use as the stencil reference value.

## Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_STENCIL_REFERENCE` dynamic state enabled

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `faceMask` **must** be a valid combination of `VkStencilFaceFlagBits` values
- `faceMask` **must** not be `0`
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

`reference` is an integer reference value that is used in the unsigned stencil comparison. The reference value used by stencil comparison must be within the range  $[0,2^s-1]$ , where  $s$  is the number of bits in the stencil framebuffer attachment, otherwise the reference value is considered undefined. The  $s$  least significant bits of `compareMask` are bitwise ANDed with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by `compareOp`. Let  $R$  be the masked reference value and  $S$  be the masked stored stencil value.

Possible values of `VkStencilOpState::compareOp`, specifying the stencil comparison function, are:

```

typedef enum VkCompareOp {
    VK_COMPARE_OP_NEVER = 0,
    VK_COMPARE_OP_LESS = 1,
    VK_COMPARE_OP_EQUAL = 2,
    VK_COMPARE_OP_LESS_OR_EQUAL = 3,
    VK_COMPARE_OP_GREATER = 4,
    VK_COMPARE_OP_NOT_EQUAL = 5,
    VK_COMPARE_OP_GREATER_OR_EQUAL = 6,
    VK_COMPARE_OP_ALWAYS = 7,
    VK_COMPARE_OP_MAX_ENUM = 0x7FFFFFFF
} VkCompareOp;

```

- **VK\_COMPARE\_OP\_NEVER** specifies that the test never passes.
- **VK\_COMPARE\_OP\_LESS** specifies that the test passes when  $R < S$ .
- **VK\_COMPARE\_OP\_EQUAL** specifies that the test passes when  $R = S$ .
- **VK\_COMPARE\_OP\_LESS\_OR\_EQUAL** specifies that the test passes when  $R \leq S$ .
- **VK\_COMPARE\_OP\_GREATER** specifies that the test passes when  $R > S$ .
- **VK\_COMPARE\_OP\_NOT\_EQUAL** specifies that the test passes when  $R \neq S$ .
- **VK\_COMPARE\_OP\_GREATER\_OR\_EQUAL** specifies that the test passes when  $R \geq S$ .
- **VK\_COMPARE\_OP\_ALWAYS** specifies that the test always passes.

Possible values of the `failOp`, `passOp`, and `depthFailOp` members of `VkStencilOpState`, specifying what happens to the stored stencil value if this or certain subsequent tests fail or pass, are:

```

typedef enum VkStencilOp {
    VK_STENCIL_OP_KEEP = 0,
    VK_STENCIL_OP_ZERO = 1,
    VK_STENCIL_OP_REPLACE = 2,
    VK_STENCIL_OP_INCREMENT_AND_CLAMP = 3,
    VK_STENCIL_OP_DECREMENT_AND_CLAMP = 4,
    VK_STENCIL_OP_INVERT = 5,
    VK_STENCIL_OP_INCREMENT_AND_WRAP = 6,
    VK_STENCIL_OP_DECREMENT_AND_WRAP = 7,
    VK_STENCIL_OP_MAX_ENUM = 0x7FFFFFFF
} VkStencilOp;

```

- **VK\_STENCIL\_OP\_KEEP** keeps the current value.
- **VK\_STENCIL\_OP\_ZERO** sets the value to 0.
- **VK\_STENCIL\_OP\_REPLACE** sets the value to `reference`.
- **VK\_STENCIL\_OP\_INCREMENT\_AND\_CLAMP** increments the current value and clamps to the maximum representable unsigned value.
- **VK\_STENCIL\_OP\_DECREMENT\_AND\_CLAMP** decrements the current value and clamps to 0.
- **VK\_STENCIL\_OP\_INVERT** bitwise-inverts the current value.

- `VK_STENCIL_OP_INCREMENT_AND_WRAP` increments the current value and wraps to 0 when the maximum value would have been exceeded.
- `VK_STENCIL_OP_DECREMENT_AND_WRAP` decrements the current value and wraps to the maximum possible value when the value would go below 0.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer.

If the stencil test fails, the sample's coverage bit is cleared in the fragment. If there is no stencil framebuffer attachment, stencil modification **cannot** occur, and it is as if the stencil tests always pass.

If the stencil test passes, the `writeMask` member of the `VkStencilOpState` structures controls how the updated stencil value is written to the stencil framebuffer attachment.

The least significant  $s$  bits of `writeMask`, where  $s$  is the number of bits in the stencil framebuffer attachment, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil value in the depth/stencil attachment is written; where a 0 appears, the bit is not written. The `writeMask` value uses either the front-facing or back-facing state based on the facingness of the fragment. Fragments generated by front-facing primitives use the front mask and fragments generated by back-facing primitives use the back mask.

## 27.13. Depth Test

The depth test conditionally disables coverage of a sample based on the outcome of a comparison between the fragment's depth value at the sample location and the sample's depth value in the depth/stencil attachment at location  $(x_f, y_f)$ . The comparison is enabled or disabled with the `depthTestEnable` member of the `VkPipelineDepthStencilStateCreateInfo` structure. When disabled, the depth comparison and subsequent possible updates to the value of the depth component of the depth/stencil attachment are bypassed and the fragment is passed to the next operation. The stencil value, however, **can** be modified as indicated above as if the depth test passed. If enabled, the comparison takes place and the depth/stencil attachment value **can** subsequently be modified.

The comparison is specified with the `depthCompareOp` member of `VkPipelineDepthStencilStateCreateInfo`. Let  $z_f$  be the incoming fragment's depth value for a sample, and let  $z_a$  be the depth/stencil attachment value in memory for that sample. The depth test passes under the following conditions:

- `VK_COMPARE_OP_NEVER`: the test never passes.
- `VK_COMPARE_OP_LESS`: the test passes when  $z_f < z_a$ .
- `VK_COMPARE_OP_EQUAL`: the test passes when  $z_f = z_a$ .
- `VK_COMPARE_OP_LESS_OR_EQUAL`: the test passes when  $z_f \leq z_a$ .
- `VK_COMPARE_OP_GREATER`: the test passes when  $z_f > z_a$ .
- `VK_COMPARE_OP_NOT_EQUAL`: the test passes when  $z_f \neq z_a$ .
- `VK_COMPARE_OP_GREATER_OR_EQUAL`: the test passes when  $z_f \geq z_a$ .
- `VK_COMPARE_OP_ALWAYS`: the test always passes.

If `VkPipelineRasterizationStateCreateInfo::depthClampEnable` is enabled, before the incoming fragment's  $z_f$  is compared to  $z_a$ ,  $z_f$  is clamped to  $[min(n,f),max(n,f)]$ , where  $n$  and  $f$  are the `minDepth` and `maxDepth` depth range values of the viewport used by this fragment, respectively.

If the depth test fails, the sample's coverage bit is cleared in the fragment. The stencil value at the sample's location is updated according to the function currently in effect for depth test failure.

If the depth test passes, the sample's (possibly clamped)  $z_f$  value is conditionally written to the depth framebuffer attachment based on the `depthWriteEnable` member of `VkPipelineDepthStencilStateCreateInfo`. The value is written if `depthWriteEnable` is `VK_TRUE` and there is a depth framebuffer attachment. Otherwise, no value is written. If the depth framebuffer attachment is a fixed-point format and the depth value is outside of the `0.0` to `1.0` range the depth value is clamped between `0.0` and `1.0` inclusive before writing. The stencil value at the sample's location is updated according to the function currently in effect for depth test success.

If there is no depth framebuffer attachment, it is as if the depth test always passes.

## 27.14. Representative Fragment Test

The representative fragment test allows implementations to reduce the amount of rasterization and fragment processing work performed for each point, line, or triangle primitive. For any primitive that produces one or more fragments that pass all prior early fragment tests, the implementation **may** choose one or more “representative” fragments for processing and discard all other fragments. For draw calls rendering multiple points, lines, or triangles arranged in lists, strips, or fans, the representative fragment test is performed independently for each of those primitives. The set of fragments discarded by the representative fragment test is implementation-dependent. In some cases, the representative fragment test may not discard any fragments for a given primitive.

If the `pNext` chain of `VkGraphicsPipelineCreateInfo` includes a `VkPipelineRepresentativeFragmentTestStateCreateInfoNV` structure, then that structure includes parameters that control the representative fragment test.

The `VkPipelineRepresentativeFragmentTestStateCreateInfoNV` structure is defined as:

```
typedef struct VkPipelineRepresentativeFragmentTestStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkBool32 representativeFragmentTestEnable;
} VkPipelineRepresentativeFragmentTestStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `representativeFragmentTestEnable` controls whether the representative fragment test is enabled.

If this structure is not present, `representativeFragmentTestEnable` is considered to be `VK_FALSE`, and the representative fragment test is disabled.

If `early fragment tests` are not enabled in the active fragment shader, the representative fragment

shader test has no effect, even if enabled.

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PIPELINE_REPRESENTATIVE_FRAGMENT_TEST_STATE_CREATE_INFO_NV`

## 27.15. Sample Counting

Occlusion queries use query pool entries to track the number of samples that pass all the per-fragment tests. The mechanism of collecting an occlusion query value is described in [Occlusion Queries](#).

The occlusion query sample counter increments by one for each sample with a coverage value of 1 in each fragment that survives all the per-fragment tests, including scissor, exclusive scissor, sample mask, alpha to coverage, stencil, and depth tests.

## 27.16. Fragment Coverage To Color

If the `pNext` chain of `VkPipelineMultisampleStateCreateInfo` includes a `VkPipelineCoverageToColorStateCreateInfoNV` structure, then that structure controls whether the fragment coverage is substituted for a fragment color output and, if so, which output is replaced.

The `VkPipelineCoverageToColorStateCreateInfoNV` structure is defined as:

```
typedef struct VkPipelineCoverageToColorStateCreateInfoNV {  
    VkStructureType sType;  
    const void* pNext;  
    VkPipelineCoverageToColorStateCreateInfoFlagsNV flags;  
    VkBool32 coverageToColorEnable;  
    uint32_t coverageToColorLocation;  
} VkPipelineCoverageToColorStateCreateInfoNV;
```

- `sType` is the type of this structure
- `pNext` is `NULL` or a pointer to an extension-specific structure
- `flags` is reserved for future use.
- `coverageToColorEnable` controls whether the fragment coverage value replaces a fragment color output.
- `coverageToColorLocation` controls which fragment shader color output value is replaced.

If `coverageToColorEnable` is `VK_TRUE`, the fragment coverage information is treated as a bitmask with one bit for each sample (as in the [Sample Mask](#) section), and this bitmask replaces the first component of the color value corresponding to the fragment shader output location with `Location` equal to `coverageToColorLocation` and `Index` equal to zero. If the color attachment format has fewer bits than the sample coverage, the low bits of the sample coverage bitmask are taken without any

clamping. If the color attachment format has more bits than the sample coverage, the high bits of the sample coverage bitmask are filled with zeros.

If [Sample Shading](#) is in use, the coverage bitmask only has bits set for samples that correspond to the fragment shader invocation that shades those samples.

This pipeline stage occurs after sample counting and before blending, and is always performed after fragment shading regardless of the setting of [EarlyFragmentTests](#).

If `coverageToColorEnable` is `VK_FALSE`, these operations are skipped. If this structure is not present, it is as if `coverageToColorEnable` is `VK_FALSE`.

## Valid Usage

- If `coverageToColorEnable` is `VK_TRUE`, then the render pass subpass indicated by `VkGraphicsPipelineCreateInfo::renderPass` and `VkGraphicsPipelineCreateInfo::subpass` **must** have a color attachment at the location selected by `coverageToColorLocation`, with a `VkFormat` of `VK_FORMAT_R8_UINT`, `VK_FORMAT_R8_SINT`, `VK_FORMAT_R16_UINT`, `VK_FORMAT_R16_SINT`, `VK_FORMAT_R32_UINT`, or `VK_FORMAT_R32_SINT`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_TO_COLOR_STATE_CREATE_INFO_NV`
- `flags` **must** be `0`

```
typedef VkFlags VkPipelineCoverageToColorStateCreateFlagsNV;
```

`VkPipelineCoverageToColorStateCreateFlagsNV` is a bitmask type for setting a mask, but is currently reserved for future use.

## 27.17. Coverage Reduction

Coverage reduction generates a *color sample mask* from the coverage mask, with one bit for each sample in the color attachment(s) for the subpass. If a bit in the color sample mask is 0, then blending and writing to the framebuffer are not performed for that sample.

When the `VK_NV_framebuffer_mixed_samples` extension is not enabled, each color sample is associated with a unique rasterization sample, and the value of the coverage mask is assigned to the color sample mask.

If the render pass has a fragment density map attachment, `rasterizationSamples` is greater than 1, and the fragment area covers multiple pixels; there is an implementation-dependent association of rasterization samples to color attachment samples within the fragment. Each color sample's mask bit is assigned the union of the coverage bits of its associated raster samples.

If the pipeline's `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` is greater than the `VkAttachmentDescription::samples` of the color attachments in the subpass, then the fragment's coverage is reduced from `rasterizationSamples` bits to a color sample mask with `VkAttachmentDescription::samples` bits.

When the `VK_NV_coverage_reduction_mode` extension is enabled, the pipeline state controlling coverage reduction is specified through the members of the `VkPipelineCoverageReductionStateCreateInfoNV` structure.

The `VkPipelineCoverageReductionStateCreateInfoNV` structure is defined as:

```
typedef struct VkPipelineCoverageReductionStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCoverageReductionStateCreateFlagsNV flags;
    VkCoverageReductionModeNV coverageReductionMode;
} VkPipelineCoverageReductionStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `coverageReductionMode` is a `VkCoverageReductionModeNV` value controlling how the *color sample mask* is generated from the coverage mask.

If this structure is not present, the default coverage reduction mode is inferred as follows:

- If the `VK_NV_framebuffer_mixed_samples` extension is enabled, then it is as if the `coverageReductionMode` is `VK_COVERAGE_REDUCTION_MODE_MERGE_NV`.
- If the `VK_AMD_mixed_attachment_samples` extension is enabled, then it is as if the `coverageReductionMode` is `VK_COVERAGE_REDUCTION_MODE_TRUNCATE_NV`.
- If both `VK_NV_framebuffer_mixed_samples` and `VK_AMD_mixed_attachment_samples` are enabled, then the default coverage reduction mode is implementation-dependent.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_REDUCTION_STATE_CREATE_INFO_NV`
- `flags` **must** be `0`
- `coverageReductionMode` **must** be a valid `VkCoverageReductionModeNV` value

```
typedef VkFlags VkPipelineCoverageReductionStateCreateFlagsNV;
```

`VkPipelineCoverageReductionStateCreateFlagsNV` is a bitmask type for setting a mask, but is currently reserved for future use.

Possible values of `VkPipelineCoverageReductionStateCreateInfoNV::coverageReductionMode`, specifying how the coverage mask is reduced to *color sample mask*, are:

```
typedef enum VkCoverageReductionModeNV {
    VK_COVERAGE_REDUCTION_MODE_MERGE_NV = 0,
    VK_COVERAGE_REDUCTION_MODE_TRUNCATE_NV = 1,
    VK_COVERAGE_REDUCTION_MODE_MAX_ENUM_NV = 0x7FFFFFFF
} VkCoverageReductionModeNV;
```

- **VK\_COVERAGE\_REDUCTION\_MODE\_MERGE\_NV**: In this mode, there is an implementation-dependent association of each raster sample to a color sample. The reduced color sample mask is computed such that the bit for each color sample is 1 if any of the associated bits in the fragment's coverage is on, and 0 otherwise.
- **VK\_COVERAGE\_REDUCTION\_MODE\_TRUNCATE\_NV**: In this mode, only the first M raster samples are associated with the color samples such that raster sample i maps to color sample i, where M is the number of color samples.

If the `VK_NV_coverage_reduction_mode` extension is not enabled, there is an implementation-dependent association of raster samples to color samples. The reduced color sample mask is computed such that the bit for each color sample is 1 if any of the associated bits in the fragment's coverage is on, and 0 otherwise.

To query the set of mixed sample combinations of coverage reduction mode, rasterization samples and color, depth, stencil attachment sample counts that are supported by a physical device, call:

```
VkResult vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV(
    VkPhysicalDevice                                     physicalDevice,
    uint32_t*                                            pCombinationCount,
    VkFramebufferMixedSamplesCombinationNV*            pCombinations);
```

- `physicalDevice` is the physical device from which to query the set of combinations.
- `pCombinationCount` is a pointer to an integer related to the number of combinations available or queried, as described below.
- `pCombinations` is either `NULL` or a pointer to an array of `VkFramebufferMixedSamplesCombinationNV` values, indicating the supported combinations of coverage reduction mode, rasterization samples, and color, depth, stencil attachment sample counts.

If `pCombinations` is `NULL`, then the number of supported combinations for the given `physicalDevice` is returned in `pCombinationCount`. Otherwise, `pCombinationCount` **must** point to a variable set by the user to the number of elements in the `pCombinations` array, and on return the variable is overwritten with the number of values actually written to `pCombinations`. If the value of `pCombinationCount` is less than the number of combinations supported for the given `physicalDevice`, at most `pCombinationCount` values will be written `pCombinations` and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the supported values were returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pCombinationCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pCombinationCount` is not `0`, and `pCombinations` is not `NULL`, `pCombinations` **must** be a valid pointer to an array of `pCombinationCount` `VkFramebufferMixedSamplesCombinationNV` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkFramebufferMixedSamplesCombinationNV` structure is defined as:

```
typedef struct VkFramebufferMixedSamplesCombinationNV {
    VkStructureType           sType;
    void*                     pNext;
    VkCoverageReductionModeNV coverageReductionMode;
    VkSampleCountFlagBits     rasterizationSamples;
    VkSampleCountFlags        depthStencilSamples;
    VkSampleCountFlags        colorSamples;
} VkFramebufferMixedSamplesCombinationNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `coverageReductionMode` is a `VkCoverageReductionModeNV` value specifying the coverage reduction mode.
- `rasterizationSamples` specifies the number of rasterization samples in the supported combination.
- `depthStencilSamples` specifies the number of samples in the depth stencil attachment in the supported combination. A value of 0 indicates the combination does not have a depth stencil attachment.
- `colorSamples` specifies the number of color samples in a color attachment in the supported combination. A value of 0 indicates the combination does not have a color attachment.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_FRAMEBUFFER_MIXED_SAMPLES_COMBINATION_NV`
- `pNext` must be `NULL`

### 27.17.1. Coverage Modulation

As part of coverage reduction, fragment color values **can** also be modulated (multiplied) by a value that is a function of fraction of covered rasterization samples associated with that color sample.

Pipeline state controlling coverage modulation is specified through the members of the `VkPipelineCoverageModulationStateCreateInfoNV` structure.

The `VkPipelineCoverageModulationStateCreateInfoNV` structure is defined as:

```
typedef struct VkPipelineCoverageModulationStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCoverageModulationStateCreateFlagsNV flags;
    VkCoverageModulationModeNV coverageModulationMode;
    VkBool32 coverageModulationTableEnable;
    uint32_t coverageModulationTableCount;
    const float* pCoverageModulationTable;
} VkPipelineCoverageModulationStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `coverageModulationMode` is a `VkCoverageModulationModeNV` value controlling which color components are modulated.
- `coverageModulationTableEnable` controls whether the modulation factor is looked up from a table in `pCoverageModulationTable`.
- `coverageModulationTableCount` is the number of elements in `pCoverageModulationTable`.
- `pCoverageModulationTable` is a table of modulation factors containing a value for each number of covered samples.

If `coverageModulationTableEnable` is `VK_FALSE`, then for each color sample the associated bits of the fragment's coverage are counted and divided by the number of associated bits to produce a modulation factor  $R$  in the range  $(0,1]$  (a value of zero would have been killed due to a color coverage of 0). Specifically:

- $N = \text{value of } \text{rasterizationSamples}$
- $M = \text{value of } \text{VkAttachmentDescription::samples}$  for any color attachments
- $R = \text{popcount}(\text{associated coverage bits}) / (N / M)$

If `coverageModulationTableEnable` is `VK_TRUE`, the value R is computed using a programmable lookup table. The lookup table has N / M elements, and the element of the table is selected by:

- $R = \text{pCoverageModulationTable}[\text{popcount}(\text{associated coverage bits}) - 1]$

Note that the table does not have an entry for  $\text{popcount}(\text{associated coverage bits}) = 0$ , because such samples would have been killed.

The values of `pCoverageModulationTable` **may** be rounded to an implementation-dependent precision, which is at least as fine as  $1 / N$ , and clamped to [0,1].

For each color attachment with a floating point or normalized color format, each fragment output color value is replicated to M values which **can** each be modulated (multiplied) by that color sample's associated value of R. Which components are modulated is controlled by `coverageModulationMode`.

If this structure is not present, it is as if `coverageModulationMode` is `VK_COVERAGE_MODULATION_MODE_NONE_NV`.

If the `coverage reduction mode` is `VK_COVERAGE_REDUCTION_MODE_TRUNCATE_NV`, each color sample is associated with only a single coverage sample. In this case, it is as if `coverageModulationMode` is `VK_COVERAGE_MODULATION_MODE_NONE_NV`.

## Valid Usage

- If `coverageModulationTableEnable` is `VK_TRUE`, `coverageModulationTableCount` **must** be equal to the number of rasterization samples divided by the number of color samples in the subpass.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_MODULATION_STATE_CREATE_INFO_NV`
- `flags` **must** be 0
- `coverageModulationMode` **must** be a valid `VkCoverageModulationModeNV` value

```
typedef VkFlags VkPipelineCoverageModulationStateCreateInfoNV;
```

`VkPipelineCoverageModulationStateCreateInfoNV` is a bitmask type for setting a mask, but is currently reserved for future use.

Possible values of `VkPipelineCoverageModulationStateCreateInfoNV::coverageModulationMode`, specifying which color components are modulated, are:

```
typedef enum VkCoverageModulationModeNV {
    VK_COVERAGE_MODULATION_MODE_NONE_NV = 0,
    VK_COVERAGE_MODULATION_MODE_RGB_NV = 1,
    VK_COVERAGE_MODULATION_MODE_ALPHA_NV = 2,
    VK_COVERAGE_MODULATION_MODE_RGBA_NV = 3,
    VK_COVERAGE_MODULATION_MODE_MAX_ENUM_NV = 0x7FFFFFFF
} VkCoverageModulationModeNV;
```

- `VK_COVERAGE_MODULATION_MODE_NONE_NV` specifies that no components are multiplied by the modulation factor.
- `VK_COVERAGE_MODULATION_MODE_RGB_NV` specifies that the red, green, and blue components are multiplied by the modulation factor.
- `VK_COVERAGE_MODULATION_MODE_ALPHA_NV` specifies that the alpha component is multiplied by the modulation factor.
- `VK_COVERAGE_MODULATION_MODE_RGBA_NV` specifies that all components are multiplied by the modulation factor.

# Chapter 28. The Framebuffer

## 28.1. Blending

Blending combines the incoming *source* fragment's R, G, B, and A values with the *destination* R, G, B, and A values of each sample stored in the framebuffer at the fragment's ( $x_f, y_f$ ) location. Blending is performed for each color sample covered by the fragment, rather than just once for each fragment.

Source and destination values are combined according to the [blend operation](#), quadruplets of source and destination weighting factors determined by the [blend factors](#), and a [blend constant](#), to obtain a new set of R, G, B, and A values, as described below.

Blending is computed and applied separately to each color attachment used by the subpass, with separate controls for each attachment.

Prior to performing the blend operation, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point as specified by [Conversion from Normalized Fixed-Point to Floating-Point](#). Blending computations are treated as if carried out in floating-point, and basic blend operations are performed with a precision and dynamic range no lower than that used to represent destination components. [Advanced blending operations](#) are performed with a precision and dynamic range no lower than the smaller of that used to represent destination components or that used to represent 16-bit floating-point values.

Blending applies only to fixed-point and floating-point color attachments. If the color attachment has an integer format, blending is not applied.

The pipeline blend state is included in the `VkPipelineColorBlendStateCreateInfo` structure during graphics pipeline creation:

The `VkPipelineColorBlendStateCreateInfo` structure is defined as:

```
typedef struct VkPipelineColorBlendStateCreateInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkPipelineColorBlendStateCreateFlags flags;
    VkBool32                  logicOpEnable;
    VkLogicOp                 logicOp;
    uint32_t                  attachmentCount;
    const VkPipelineColorBlendAttachmentState* pAttachments;
    float                      blendConstants[4];
} VkPipelineColorBlendStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `logicOpEnable` controls whether to apply [Logical Operations](#).

- `logicOp` selects which logical operation to apply.
- `attachmentCount` is the number of `VkPipelineColorBlendAttachmentState` elements in `pAttachments`. This value **must** equal the `colorAttachmentCount` for the subpass in which this pipeline is used.
- `pAttachments`: is a pointer to an array of per target attachment states.
- `blendConstants` is a pointer to an array of four values used as the R, G, B, and A components of the blend constant that are used in blending, depending on the `blend factor`.

Each element of the `pAttachments` array is a `VkPipelineColorBlendAttachmentState` structure specifying per-target blending state for each individual color attachment. If the `independent blending` feature is not enabled on the device, all `VkPipelineColorBlendAttachmentState` elements in the `pAttachments` array **must** be identical.

## Valid Usage

- If the `independent blending` feature is not enabled, all elements of `pAttachments` **must** be identical
- If the `logic operations` feature is not enabled, `logicOpEnable` **must** be `VK_FALSE`
- If `logicOpEnable` is `VK_TRUE`, `logicOp` **must** be a valid `VkLogicOp` value

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkPipelineColorBlendAdvancedStateCreateInfoEXT`
- `flags` **must** be `0`
- If `attachmentCount` is not `0`, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkPipelineColorBlendAttachmentState` structures

```
typedef VkFlags VkPipelineColorBlendStateCreateFlags;
```

`VkPipelineColorBlendStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The `VkPipelineColorBlendAttachmentState` structure is defined as:

```

typedef struct VkPipelineColorBlendAttachmentState {
    VkBool32                blendEnable;
    VkBlendFactor            srcColorBlendFactor;
    VkBlendFactor            dstColorBlendFactor;
    VkBlendOp                colorBlendOp;
    VkBlendFactor            srcAlphaBlendFactor;
    VkBlendFactor            dstAlphaBlendFactor;
    VkBlendOp                alphaBlendOp;
    VkColorComponentFlags    colorWriteMask;
} VkPipelineColorBlendAttachmentState;

```

- **blendEnable** controls whether blending is enabled for the corresponding color attachment. If blending is not enabled, the source fragment's color for that attachment is passed through unmodified.
- **srcColorBlendFactor** selects which blend factor is used to determine the source factors ( $S_r, S_g, S_b$ ).
- **dstColorBlendFactor** selects which blend factor is used to determine the destination factors ( $D_r, D_g, D_b$ ).
- **colorBlendOp** selects which blend operation is used to calculate the RGB values to write to the color attachment.
- **srcAlphaBlendFactor** selects which blend factor is used to determine the source factor  $S_a$ .
- **dstAlphaBlendFactor** selects which blend factor is used to determine the destination factor  $D_a$ .
- **alphaBlendOp** selects which blend operation is used to calculate the alpha values to write to the color attachment.
- **colorWriteMask** is a bitmask of [VkColorComponentFlagBits](#) specifying which of the R, G, B, and/or A components are enabled for writing, as described for the [Color Write Mask](#).

## Valid Usage

- If the `dual source blending` feature is not enabled, `srcColorBlendFactor` **must** not be `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`
- If the `dual source blending` feature is not enabled, `dstColorBlendFactor` **must** not be `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`
- If the `dual source blending` feature is not enabled, `srcAlphaBlendFactor` **must** not be `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`
- If the `dual source blending` feature is not enabled, `dstAlphaBlendFactor` **must** not be `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`
- If either of `colorBlendOp` or `alphaBlendOp` is an `advanced blend operation`, then `colorBlendOp` **must** equal `alphaBlendOp`
- If `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendIndependentBlend` is `VK_FALSE` and `colorBlendOp` is an `advanced blend operation`, then `colorBlendOp` **must** be the same for all attachments.
- If `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendIndependentBlend` is `VK_FALSE` and `alphaBlendOp` is an `advanced blend operation`, then `alphaBlendOp` **must** be the same for all attachments.
- If `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendAllOperations` is `VK_FALSE`, then `colorBlendOp` **must** not be `VK_BLEND_OP_ZERO_EXT`, `VK_BLEND_OP_SRC_EXT`, `VK_BLEND_OP_DST_EXT`, `VK_BLEND_OP_SRC_OVER_EXT`, `VK_BLEND_OP_DST_OVER_EXT`, `VK_BLEND_OP_SRC_IN_EXT`, `VK_BLEND_OP_DST_IN_EXT`, `VK_BLEND_OP_SRC_OUT_EXT`, `VK_BLEND_OP_DST_OUT_EXT`, `VK_BLEND_OP_SRC_ATOP_EXT`, `VK_BLEND_OP_DST_ATOP_EXT`, `VK_BLEND_OP_XOR_EXT`, `VK_BLEND_OP_INVERT_EXT`, `VK_BLEND_OP_INVERT_RGB_EXT`, `VK_BLEND_OP_LINEAR_DODGE_EXT`, `VK_BLEND_OP_LINEAR_BURN_EXT`, `VK_BLEND_OP_VIVID_LIGHT_EXT`, `VK_BLEND_OP_LINEAR_LIGHT_EXT`, `VK_BLEND_OP_PINLIGHT_EXT`, `VK_BLEND_OP_HARD_MIX_EXT`, `VK_BLEND_OP_PLUS_EXT`, `VK_BLEND_OP_PLUS_CLAMPED_EXT`, `VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT`, `VK_BLEND_OP_PLUS_DARKER_EXT`, `VK_BLEND_OP_MINUS_EXT`, `VK_BLEND_OP_MINUS_CLAMPED_EXT`, `VK_BLEND_OP_CONTRAST_EXT`, `VK_BLEND_OP_INVERT_OVG_EXT`, `VK_BLEND_OP_RED_EXT`, `VK_BLEND_OP_GREEN_EXT`, or `VK_BLEND_OP_BLUE_EXT`
- If `colorBlendOp` or `alphaBlendOp` is an `advanced blend operation`, then `VkSubpassDescription::colorAttachmentCount` of the subpass this pipeline is compiled against **must** be less than or equal to `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendMaxColorAttachments`

## Valid Usage (Implicit)

- `srcColorBlendFactor` **must** be a valid `VkBlendFactor` value
- `dstColorBlendFactor` **must** be a valid `VkBlendFactor` value
- `colorBlendOp` **must** be a valid `VkBlendOp` value
- `srcAlphaBlendFactor` **must** be a valid `VkBlendFactor` value
- `dstAlphaBlendFactor` **must** be a valid `VkBlendFactor` value
- `alphaBlendOp` **must** be a valid `VkBlendOp` value
- `colorWriteMask` **must** be a valid combination of `VkColorComponentFlagBits` values

### 28.1.1. Blend Factors

The source and destination color and alpha blending factors are selected from the enum:

```
typedef enum VkBlendFactor {  
    VK_BLEND_FACTOR_ZERO = 0,  
    VK_BLEND_FACTOR_ONE = 1,  
    VK_BLEND_FACTOR_SRC_COLOR = 2,  
    VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR = 3,  
    VK_BLEND_FACTOR_DST_COLOR = 4,  
    VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR = 5,  
    VK_BLEND_FACTOR_SRC_ALPHA = 6,  
    VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA = 7,  
    VK_BLEND_FACTOR_DST_ALPHA = 8,  
    VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA = 9,  
    VK_BLEND_FACTOR_CONSTANT_COLOR = 10,  
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR = 11,  
    VK_BLEND_FACTOR_CONSTANT_ALPHA = 12,  
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA = 13,  
    VK_BLEND_FACTOR_SRC_ALPHA_SATURATE = 14,  
    VK_BLEND_FACTOR_SRC1_COLOR = 15,  
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR = 16,  
    VK_BLEND_FACTOR_SRC1_ALPHA = 17,  
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA = 18,  
    VK_BLEND_FACTOR_MAX_ENUM = 0x7FFFFFFF  
} VkBlendFactor;
```

The semantics of each enum value is described in the table below:

*Table 33. Blend Factors*

<b>VkBlendFactor</b>	<b>RGB Blend Factors (<math>S_r, S_g, S_b</math>) or (<math>D_r, D_g, D_b</math>)</b>	<b>Alpha Blend Factor (<math>S_a</math> or <math>D_a</math>)</b>
VK_BLEND_FACTOR_ZERO	(0,0,0)	0
VK_BLEND_FACTOR_ONE	(1,1,1)	1
VK_BLEND_FACTOR_SRC_COLOR	( $R_{s0}, G_{s0}, B_{s0}$ )	$A_{s0}$
VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR	( $1-R_{s0}, 1-G_{s0}, 1-B_{s0}$ )	$1-A_{s0}$
VK_BLEND_FACTOR_DST_COLOR	( $R_d, G_d, B_d$ )	$A_d$
VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR	( $1-R_d, 1-G_d, 1-B_d$ )	$1-A_d$
VK_BLEND_FACTOR_SRC_ALPHA	( $A_{s0}, A_{s0}, A_{s0}$ )	$A_{s0}$
VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA	( $1-A_{s0}, 1-A_{s0}, 1-A_{s0}$ )	$1-A_{s0}$
VK_BLEND_FACTOR_DST_ALPHA	( $A_d, A_d, A_d$ )	$A_d$
VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA	( $1-A_d, 1-A_d, 1-A_d$ )	$1-A_d$
VK_BLEND_FACTOR_CONSTANT_COLOR	( $R_c, G_c, B_c$ )	$A_c$
VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR	( $1-R_c, 1-G_c, 1-B_c$ )	$1-A_c$
VK_BLEND_FACTOR_CONSTANT_ALPHA	( $A_c, A_c, A_c$ )	$A_c$
VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA	( $1-A_c, 1-A_c, 1-A_c$ )	$1-A_c$
VK_BLEND_FACTOR_SRC_ALPHA_SATURATE	( $f, f, f$ ); $f = \min(A_{s0}, 1-A_d)$	1
VK_BLEND_FACTOR_SRC1_COLOR	( $R_{s1}, G_{s1}, B_{s1}$ )	$A_{s1}$
VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR	( $1-R_{s1}, 1-G_{s1}, 1-B_{s1}$ )	$1-A_{s1}$
VK_BLEND_FACTOR_SRC1_ALPHA	( $A_{s1}, A_{s1}, A_{s1}$ )	$A_{s1}$
VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA	( $1-A_{s1}, 1-A_{s1}, 1-A_{s1}$ )	$1-A_{s1}$

In this table, the following conventions are used:

- $R_{s0}, G_{s0}, B_{s0}$  and  $A_{s0}$  represent the first source color R, G, B, and A components, respectively, for the fragment output location corresponding to the color attachment being blended.
- $R_{s1}, G_{s1}, B_{s1}$  and  $A_{s1}$  represent the second source color R, G, B, and A components, respectively, used in dual source blending modes, for the fragment output location corresponding to the color attachment being blended.
- $R_d, G_d, B_d$  and  $A_d$  represent the R, G, B, and A components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.
- $R_c, G_c, B_c$  and  $A_c$  represent the blend constant R, G, B, and A components, respectively.

If the pipeline state object is created without the `VK_DYNAMIC_STATE_BLEND_CONSTANTS` dynamic state enabled then the *blend constant* ( $R_c, G_c, B_c, A_c$ ) is specified via the `blendConstants` member of `VkPipelineColorBlendStateCreateInfo`.

Otherwise, to dynamically set and change the blend constant, call:

```
void vkCmdSetBlendConstants(
    VkCommandBuffer
    const float
```

commandBuffer,  
blendConstants[4]);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `blendConstants` is a pointer to an array of four values specifying the R, G, B, and A components of the blend constant color used in blending, depending on the `blend factor`.

### Valid Usage

- The bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_BLEND_CONSTANTS` dynamic state enabled

### Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

### Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics	

## 28.1.2. Dual-Source Blending

Blend factors that use the secondary color input ( $R_{s1}, G_{s1}, B_{s1}, A_{s1}$ ) (`VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, and `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`) **may** consume implementation resources that could otherwise be used for rendering to multiple color attachments. Therefore, the number of color attachments that **can** be used in a framebuffer **may** be lower when using dual-source blending.

Dual-source blending is only supported if the `dualSrcBlend` feature is enabled.

The maximum number of color attachments that **can** be used in a subpass when using dual-source blending functions is implementation-dependent and is reported as the `maxFragmentDualSrcAttachments` member of `VkPhysicalDeviceLimits`.

When using a fragment shader with dual-source blending functions, the color outputs are bound to the first and second inputs of the blender using the `Index` decoration, as described in [Fragment Output Interface](#). If the second color input to the blender is not written in the shader, or if no output is bound to the second input of a blender, the result of the blending operation is not defined.

### 28.1.3. Blend Operations

Once the source and destination blend factors have been selected, they along with the source and destination components are passed to the blending operations. RGB and alpha components **can** use different operations. Possible values of `VkBlendOp`, specifying the operations, are:

```
typedef enum VkBlendOp {
    VK_BLEND_OP_ADD = 0,
    VK_BLEND_OP_SUBTRACT = 1,
    VK_BLEND_OP_REVERSE_SUBTRACT = 2,
    VK_BLEND_OP_MIN = 3,
    VK_BLEND_OP_MAX = 4,
    VK_BLEND_OP_ZERO_EXT = 1000148000,
    VK_BLEND_OP_SRC_EXT = 1000148001,
    VK_BLEND_OP_DST_EXT = 1000148002,
    VK_BLEND_OP_SRC_OVER_EXT = 1000148003,
    VK_BLEND_OP_DST_OVER_EXT = 1000148004,
    VK_BLEND_OP_SRC_IN_EXT = 1000148005,
    VK_BLEND_OP_DST_IN_EXT = 1000148006,
    VK_BLEND_OP_SRC_OUT_EXT = 1000148007,
    VK_BLEND_OP_DST_OUT_EXT = 1000148008,
    VK_BLEND_OP_SRC_ATOP_EXT = 1000148009,
    VK_BLEND_OP_DST_ATOP_EXT = 1000148010,
    VK_BLEND_OP_XOR_EXT = 1000148011,
    VK_BLEND_OP_MULTIPLY_EXT = 1000148012,
    VK_BLEND_OP_SCREEN_EXT = 1000148013,
    VK_BLEND_OP_OVERLAY_EXT = 1000148014,
    VK_BLEND_OP_DARKEN_EXT = 1000148015,
    VK_BLEND_OP_LIGHTEN_EXT = 1000148016,
    VK_BLEND_OP_COLORDODGE_EXT = 1000148017,
    VK_BLEND_OP_COLORBURN_EXT = 1000148018,
    VK_BLEND_OP_HARDLIGHT_EXT = 1000148019,
    VK_BLEND_OP_SOFTLIGHT_EXT = 1000148020,
    VK_BLEND_OP_DIFFERENCE_EXT = 1000148021,
    VK_BLEND_OP_EXCLUSION_EXT = 1000148022,
    VK_BLEND_OP_INVERT_EXT = 1000148023,
    VK_BLEND_OP_INVERT_RGB_EXT = 1000148024,
    VK_BLEND_OP_LINEARDODGE_EXT = 1000148025,
    VK_BLEND_OP_LINEARBURN_EXT = 1000148026,
```

```
VK_BLEND_OP_VIVIDLIGHT_EXT = 1000148027,
VK_BLEND_OP_LINEARLIGHT_EXT = 1000148028,
VK_BLEND_OP_PINLIGHT_EXT = 1000148029,
VK_BLEND_OP_HARDMIX_EXT = 1000148030,
VK_BLEND_OP_HSL_HUE_EXT = 1000148031,
VK_BLEND_OP_HSL_SATURATION_EXT = 1000148032,
VK_BLEND_OP_HSL_COLOR_EXT = 1000148033,
VK_BLEND_OP_HSL_LUMINOSITY_EXT = 1000148034,
VK_BLEND_OP_PLUS_EXT = 1000148035,
VK_BLEND_OP_PLUS_CLAMPED_EXT = 1000148036,
VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT = 1000148037,
VK_BLEND_OP_PLUS_DARKER_EXT = 1000148038,
VK_BLEND_OP_MINUS_EXT = 1000148039,
VK_BLEND_OP_MINUS_CLAMPED_EXT = 1000148040,
VK_BLEND_OP_CONTRAST_EXT = 1000148041,
VK_BLEND_OP_INVERT_OVG_EXT = 1000148042,
VK_BLEND_OP_RED_EXT = 1000148043,
VK_BLEND_OP_GREEN_EXT = 1000148044,
VK_BLEND_OP_BLUE_EXT = 1000148045,
VK_BLEND_OP_MAX_ENUM = 0x7FFFFFFF
} VkBlendOp;
```

The semantics of each basic blend operations is described in the table below:

*Table 34. Basic Blend Operations*

VkBlendOp	RGB Components	Alpha Component
VK_BLEND_OP_ADD	$R = R_{s0} \times S_r + R_d \times D_r$ $G = G_{s0} \times S_g + G_d \times D_g$ $B = B_{s0} \times S_b + B_d \times D_b$	$A = A_{s0} \times S_a + A_d \times D_a$
VK_BLEND_OP_SUBTRACT	$R = R_{s0} \times S_r - R_d \times D_r$ $G = G_{s0} \times S_g - G_d \times D_g$ $B = B_{s0} \times S_b - B_d \times D_b$	$A = A_{s0} \times S_a - A_d \times D_a$
VK_BLEND_OP_REVERSE_SUBTRACT	$R = R_d \times D_r - R_{s0} \times S_r$ $G = G_d \times D_g - G_{s0} \times S_g$ $B = B_d \times D_b - B_{s0} \times S_b$	$A = A_d \times D_a - A_{s0} \times S_a$
VK_BLEND_OP_MIN	$R = \min(R_{s0}, R_d)$ $G = \min(G_{s0}, G_d)$ $B = \min(B_{s0}, B_d)$	$A = \min(A_{s0}, A_d)$
VK_BLEND_OP_MAX	$R = \max(R_{s0}, R_d)$ $G = \max(G_{s0}, G_d)$ $B = \max(B_{s0}, B_d)$	$A = \max(A_{s0}, A_d)$

In this table, the following conventions are used:

- $R_{s0}$ ,  $G_{s0}$ ,  $B_{s0}$  and  $A_{s0}$  represent the first source color R, G, B, and A components, respectively.
- $R_d$ ,  $G_d$ ,  $B_d$  and  $A_d$  represent the R, G, B, and A components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.
- $S_r$ ,  $S_g$ ,  $S_b$  and  $S_a$  represent the source blend factor R, G, B, and A components, respectively.
- $D_r$ ,  $D_g$ ,  $D_b$  and  $D_a$  represent the destination blend factor R, G, B, and A components, respectively.

The blending operation produces a new set of values R, G, B and A, which are written to the framebuffer attachment. If blending is not enabled for this attachment, then R, G, B and A are assigned  $R_{s0}$ ,  $G_{s0}$ ,  $B_{s0}$  and  $A_{s0}$ , respectively.

If the color attachment is fixed-point, the components of the source and destination values and blend factors are each clamped to [0,1] or [-1,1] respectively for an unsigned normalized or signed normalized color attachment prior to evaluating the blend operations. If the color attachment is floating-point, no clamping occurs.

If the numeric format of a framebuffer attachment uses sRGB encoding, the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence are linearized prior to their use in blending. Each R, G, and B component is converted from nonlinear to linear as described in the “sRGB EOTF” section of the [Khronos Data Format Specification](#). If the format is not sRGB, no linearization is performed.

If the numeric format of a framebuffer attachment uses sRGB encoding, then the final R, G and B values are converted into the nonlinear sRGB representation before being written to the framebuffer attachment as described in the “sRGB EOTF<sup>-1</sup>” section of the Khronos Data Format Specification.

If the framebuffer color attachment numeric format is not sRGB encoded then the resulting  $c_s$  values for R, G and B are unmodified. The value of A is never sRGB encoded. That is, the alpha component is always stored in memory as linear.

If the framebuffer color attachment is `VK_ATTACHMENT_UNUSED`, no writes are performed through that attachment. Framebuffer color attachments greater than or equal to `VkSubpassDescription::colorAttachmentCount` perform no writes.

## 28.1.4. Advanced Blend Operations

The *advanced blend operations* are those listed in tables [f/X/Y/Z Advanced Blend Operations](#), [Hue-Saturation-Luminosity Advanced Blend Operations](#), and [Additional RGB Blend Operations](#).

If the `pNext` chain of `VkPipelineColorBlendStateCreateInfo` includes a `VkPipelineColorBlendAdvancedStateCreateInfoEXT` structure, then that structure includes parameters that affect advanced blend operations.

The `VkPipelineColorBlendAdvancedStateCreateInfoEXT` structure is defined as:

```
typedef struct VkPipelineColorBlendAdvancedStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkBool32 srcPremultiplied;
    VkBool32 dstPremultiplied;
    VkBlendOverlapEXT blendOverlap;
} VkPipelineColorBlendAdvancedStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcPremultiplied` specifies whether the source color of the blend operation is treated as premultiplied.
- `dstPremultiplied` specifies whether the destination color of the blend operation is treated as premultiplied.
- `blendOverlap` is a `VkBlendOverlapEXT` value specifying how the source and destination sample's coverage is correlated.

If this structure is not present, `srcPremultiplied` and `dstPremultiplied` are both considered to be `VK_TRUE`, and `blendOverlap` is considered to be `VK_BLEND_OVERLAP_UNCORRELATED_EXT`.

## Valid Usage

- If the `non-premultiplied source color` property is not supported, `srcPremultiplied` must be `VK_TRUE`
- If the `non-premultiplied destination color` property is not supported, `dstPremultiplied` must be `VK_TRUE`
- If the `correlated overlap` property is not supported, `blendOverlap` must be `VK_BLEND_OVERLAP_UNCORRELATED_EXT`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT`
- `blendOverlap` must be a valid `VkBlendOverlapEXT` value

When using one of the operations in table [f/X/Y/Z Advanced Blend Operations](#) or [Hue-Saturation-Luminosity Advanced Blend Operations](#), blending is performed according to the following equations:

$$\begin{aligned} R &= f(R_s', R_d') * p_0(A_s, A_d) + Y * R_s' * p_1(A_s, A_d) + Z * R_d' * p_2(A_s, A_d) \\ G &= f(G_s', G_d') * p_0(A_s, A_d) + Y * G_s' * p_1(A_s, A_d) + Z * G_d' * p_2(A_s, A_d) \\ B &= f(B_s', B_d') * p_0(A_s, A_d) + Y * B_s' * p_1(A_s, A_d) + Z * B_d' * p_2(A_s, A_d) \\ A &= X * p_0(A_s, A_d) + Y * p_1(A_s, A_d) + Z * p_2(A_s, A_d) \end{aligned}$$

where the function  $f$  and terms X, Y, and Z are specified in the table. The R, G, and B components of the source color used for blending are derived according to `srcPremultiplied`. If `srcPremultiplied` is set to `VK_TRUE`, the fragment color components are considered to have been premultiplied by the A component prior to blending. The base source color  $(R_s', G_s', B_s')$  is obtained by dividing through by the A component:

$$(R_s', G_s', B_s') = \begin{cases} (0, 0, 0) & A_s = 0 \\ (\frac{R_s}{A_s}, \frac{G_s}{A_s}, \frac{B_s}{A_s}) & \text{otherwise} \end{cases}$$

If `srcPremultiplied` is `VK_FALSE`, the fragment color components are used as the base color:

$$(R_s', G_s', B_s') = (R_s, G_s, B_s)$$

The R, G, and B components of the destination color used for blending are derived according to `dstPremultiplied`. If `dstPremultiplied` is set to `VK_TRUE`, the destination components are considered to have been premultiplied by the A component prior to blending. The base destination color  $(R_d', G_d', B_d')$  is obtained by dividing through by the A component:

$$(R_d', G_d', B_d') = \begin{cases} (0, 0, 0) & A_d = 0 \\ (\frac{R_d}{A_d}, \frac{G_d}{A_d}, \frac{B_d}{A_d}) & \text{otherwise} \end{cases}$$

If `dstPremultiplied` is `VK_FALSE`, the destination color components are used as the base color:

$$(R_d', G_d', B_d') = (R_d, G_d, B_d)$$

When blending using advanced blend operations, we expect that the R, G, and B components of premultiplied source and destination color inputs be stored as the product of non-premultiplied R, G, and B component values and the A component of the color. If any R, G, or B component of a premultiplied input color is non-zero and the A component is zero, the color is considered ill-formed, and the corresponding component of the blend result is undefined.

All of the advanced blend operation formulas in this chapter compute the result as a premultiplied color. If `dstPremultiplied` is `VK_FALSE`, that result color's R, G, and B components are divided by the A component before being written to the framebuffer. If any R, G, or B component of the color is non-zero and the A component is zero, the result is considered ill-formed, and the corresponding component of the blend result is undefined. If all components are zero, that value is unchanged.

If the A component of any input or result color is less than zero, the color is considered ill-formed, and all components of the blend result are undefined.

The weighting functions  $p_0$ ,  $p_1$ , and  $p_2$  are defined in table [Advanced Blend Overlap Modes](#). In these functions, the A components of the source and destination colors are taken to indicate the portion of the pixel covered by the fragment (source) and the fragments previously accumulated in the pixel (destination). The functions  $p_0$ ,  $p_1$ , and  $p_2$  approximate the relative portion of the pixel covered by the intersection of the source and destination, covered only by the source, and covered only by the destination, respectively.

Possible values of `VkPipelineColorBlendAdvancedStateCreateInfoEXT::blendOverlap`, specifying the blend overlap functions, are:

```
typedef enum VkBlendOverlapEXT {
    VK_BLEND_OVERLAP_UNCORRELATED_EXT = 0,
    VK_BLEND_OVERLAP_DISJOINT_EXT = 1,
    VK_BLEND_OVERLAP_CONJOINT_EXT = 2,
    VK_BLEND_OVERLAP_MAX_ENUM_EXT = 0x7FFFFFFF
} VkBlendOverlapEXT;
```

- `VK_BLEND_OVERLAP_UNCORRELATED_EXT` specifies that there is no correlation between the source and destination coverage.
- `VK_BLEND_OVERLAP_CONJOINT_EXT` specifies that the source and destination coverage are considered to have maximal overlap.
- `VK_BLEND_OVERLAP_DISJOINT_EXT` specifies that the source and destination coverage are considered to have minimal overlap.

*Table 35. Advanced Blend Overlap Modes*

Overlap Mode	Weighting Equations
<code>VK_BLEND_OVERLAP_UNCORRELATED_EXT</code>	$p_0(A_s, A_d) = A_s A_d$ $p_1(A_s, A_d) = A_s(1 - A_d)$ $p_2(A_s, A_d) = A_d(1 - A_s)$

Overlap Mode	Weighting Equations
<code>VK_BLEND_OVERLAP_CONJOINT_EXT</code>	$p_0(A_s, A_d) = \min(A_s, A_d)$ $p_1(A_s, A_d) = \max(A_s - A_d, 0)$ $p_2(A_s, A_d) = \max(A_d - A_s, 0)$
<code>VK_BLEND_OVERLAP_DISJOINT_EXT</code>	$p_0(A_s, A_d) = \max(A_s + A_d - 1, 0)$ $p_1(A_s, A_d) = \min(A_s, 1 - A_d)$ $p_2(A_s, A_d) = \min(A_d, 1 - A_s)$

Table 36. f/X/Y/Z Advanced Blend Operations

Mode	Blend Coefficients
<code>VK_BLEND_OP_ZERO_EXT</code>	$(X, Y, Z) = (0, 0, 0)$ $f(C_s, C_d) = 0$
<code>VK_BLEND_OP_SRC_EXT</code>	$(X, Y, Z) = (1, 1, 0)$ $f(C_s, C_d) = C_s$
<code>VK_BLEND_OP_DST_EXT</code>	$(X, Y, Z) = (1, 0, 1)$ $f(C_s, C_d) = C_d$
<code>VK_BLEND_OP_SRC_OVER_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = C_s$
<code>VK_BLEND_OP_DST_OVER_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = C_d$
<code>VK_BLEND_OP_SRC_IN_EXT</code>	$(X, Y, Z) = (1, 0, 0)$ $f(C_s, C_d) = C_s$
<code>VK_BLEND_OP_DST_IN_EXT</code>	$(X, Y, Z) = (1, 0, 0)$ $f(C_s, C_d) = C_d$
<code>VK_BLEND_OP_SRC_OUT_EXT</code>	$(X, Y, Z) = (0, 1, 0)$ $f(C_s, C_d) = 0$
<code>VK_BLEND_OP_DST_OUT_EXT</code>	$(X, Y, Z) = (0, 0, 1)$ $f(C_s, C_d) = 0$
<code>VK_BLEND_OP_SRC_ATOP_EXT</code>	$(X, Y, Z) = (1, 0, 1)$ $f(C_s, C_d) = C_s$
<code>VK_BLEND_OP_DST_ATOP_EXT</code>	$(X, Y, Z) = (1, 1, 0)$ $f(C_s, C_d) = C_d$
<code>VK_BLEND_OP_XOR_EXT</code>	$(X, Y, Z) = (0, 1, 1)$ $f(C_s, C_d) = 0$

Mode	Blend Coefficients
<code>VK_BLEND_OP_MULTIPLY_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = C_s C_d$
<code>VK_BLEND_OP_SCREEN_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = C_s + C_d - C_s C_d$
<code>VK_BLEND_OP_OVERLAY_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 2C_s C_d & C_d \leq 0.5 \\ 1 - 2(1 - C_s)(1 - C_d) & \text{otherwise} \end{cases}$
<code>VK_BLEND_OP_DARKEN_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \min(C_s, C_d)$
<code>VK_BLEND_OP_LIGHTEN_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \max(C_s, C_d)$
<code>VK_BLEND_OP_COLORDODGE_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 0 & C_d \leq 0 \\ \min(1, \frac{C_d}{1 - C_s}) & C_d > 0 \text{ and } C_s < 1 \\ 1 & C_d > 0 \text{ and } C_s \geq 1 \end{cases}$
<code>VK_BLEND_OP_COLORBURN_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 1 & C_d \geq 1 \\ 1 - \min(1, \frac{1 - C_d}{C_s}) & C_d < 1 \text{ and } C_s > 0 \\ 0 & C_d < 1 \text{ and } C_s \leq 0 \end{cases}$
<code>VK_BLEND_OP_HARDLIGHT_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 2C_s C_d & C_s \leq 0.5 \\ 1 - 2(1 - C_s)(1 - C_d) & \text{otherwise} \end{cases}$
<code>VK_BLEND_OP_SOFTLIGHT_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} C_d - (1 - 2C_s)C_d(1 - C_d) & C_s \leq 0.5 \\ C_d + (2C_s - 1)C_d((16C_d - 12)C_d + 3) & C_s > 0.5 \text{ and } C_d \leq 0.25 \\ C_d + (2C_s - 1)(\sqrt{C_d} - C_d) & C_s > 0.5 \text{ and } C_d > 0.25 \end{cases}$
<code>VK_BLEND_OP_DIFFERENCE_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) =  C_d - C_s $
<code>VK_BLEND_OP_EXCLUSION_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = C_s + C_d - 2C_s C_d$
<code>VK_BLEND_OP_INVERT_EXT</code>	$(X, Y, Z) = (1, 0, 1)$ $f(C_s, C_d) = 1 - C_d$
<code>VK_BLEND_OP_INVERT_RGB_EXT</code>	$(X, Y, Z) = (1, 0, 1)$ $f(C_s, C_d) = C_s(1 - C_d)$

Mode	Blend Coefficients
<code>VK_BLEND_OP_LINEARDODGE_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} C_s + C_d & C_s + C_d \leq 1 \\ 1 & \text{otherwise} \end{cases}$
<code>VK_BLEND_OP_LINEARBURN_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} C_s + C_d - 1 & C_s + C_d > 1 \\ 0 & \text{otherwise} \end{cases}$
<code>VK_BLEND_OP_VIVIDLIGHT_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 1 - \min(1, \frac{1 - C_d}{2C_s}) & 0 < C_s < 0.5 \\ 0 & C_s \leq 0 \\ \min(1, \frac{C_d}{2(1 - C_s)}) & 0.5 \leq C_s < 1 \\ 1 & C_s \geq 1 \end{cases}$
<code>VK_BLEND_OP_LINEARLIGHT_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 1 & 2C_s + C_d > 2 \\ 2C_s + C_d - 1 & 1 < 2C_s + C_d \leq 2 \\ 0 & 2C_s + C_d \leq 1 \end{cases}$
<code>VK_BLEND_OP_PINLIGHT_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 0 & 2C_s - 1 > C_d \text{ and } C_s < 0.5 \\ 2C_s - 1 & 2C_s - 1 > C_d \text{ and } C_s \geq 0.5 \\ 2C_s & 2C_s - 1 \leq C_d \text{ and } C_s < 0.5C_d \\ C_d & 2C_s - 1 \leq C_d \text{ and } C_s \geq 0.5C_d \end{cases}$
<code>VK_BLEND_OP_HARDMIX_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = \begin{cases} 0 & C_s + C_d < 1 \\ 1 & \text{otherwise} \end{cases}$

When using one of the HSL blend operations in table [Hue-Saturation-Luminosity Advanced Blend Operations](#) as the blend operation, the RGB color components produced by the function  $f$  are effectively obtained by converting both the non-premultiplied source and destination colors to the HSL (hue, saturation, luminosity) color space, generating a new HSL color by selecting H, S, and L components from the source or destination according to the blend operation, and then converting the result back to RGB. In the equations below, a blended RGB color is produced according to the following pseudocode:

```

float minv3(vec3 c) {
    return min(min(c.r, c.g), c.b);
}
float maxv3(vec3 c) {
    return max(max(c.r, c.g), c.b);
}
float lumv3(vec3 c) {
    return dot(c, vec3(0.30, 0.59, 0.11));
}
float satv3(vec3 c) {

```

```

    return maxv3(c) - minv3(c);
}

// If any color components are outside [0,1], adjust the color to
// get the components in range.
vec3 ClipColor(vec3 color) {
    float lum = lumv3(color);
    float mincol = minv3(color);
    float maxcol = maxv3(color);
    if (mincol < 0.0) {
        color = lum + ((color-lum)*lum) / (lum-mincol);
    }
    if (maxcol > 1.0) {
        color = lum + ((color-lum)*(1-lum)) / (maxcol-lum);
    }
    return color;
}

// Take the base RGB color <cbase> and override its luminosity
// with that of the RGB color <clum>.
vec3 SetLum(vec3 cbase, vec3 clum) {
    float lbase = lumv3(cbase);
    float llum = lumv3(clum);
    float ldiff = llum - lbase;
    vec3 color = cbase + vec3(ldiff);
    return ClipColor(color);
}

// Take the base RGB color <cbase> and override its saturation with
// that of the RGB color <csat>. The override the luminosity of the
// result with that of the RGB color <clum>.
vec3 SetLumSat(vec3 cbase, vec3 csat, vec3 clum)
{
    float minbase = minv3(cbase);
    float sbase = satv3(cbase);
    float ssat = satv3(csat);
    vec3 color;
    if (sbase > 0) {
        // Equivalent (modulo rounding errors) to setting the
        // smallest (R,G,B) component to 0, the largest to <ssat>,
        // and interpolating the "middle" component based on its
        // original value relative to the smallest/largest.
        color = (cbase - minbase) * ssat / sbase;
    } else {
        color = vec3(0.0);
    }
    return SetLum(color, clum);
}

```

*Table 37. Hue-Saturation-Luminosity Advanced Blend Operations*

Mode	Result
<code>VK_BLEND_OP_HSL_HUE_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = SetLumSat(C_s, C_d, C_d)$
<code>VK_BLEND_OP_HSL_SATURATION_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = SetLumSat(C_d, C_s, C_d)$
<code>VK_BLEND_OP_HSL_COLOR_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = SetLum(C_s, C_d)$
<code>VK_BLEND_OP_HSL_LUMINOSITY_EXT</code>	$(X, Y, Z) = (1, 1, 1)$ $f(C_s, C_d) = SetLum(C_d, C_s)$

When using one of the operations in table [Additional RGB Blend Operations](#) as the blend operation, the source and destination colors used by these blending operations are interpreted according to `srcPremultiplied` and `dstPremultiplied`. The blending operations below are evaluated where the RGB source and destination color components are both considered to have been premultiplied by the corresponding A component.

$$(R_s', G_s', B_s') = \begin{cases} (R_s, G_s, B_s) & \text{if } \text{srcPremultiplied} \text{ is VK\_TRUE} \\ (R_s A_s, G_s A_s, B_s A_s) & \text{if } \text{srcPremultiplied} \text{ is VK\_FALSE} \end{cases}$$

$$(R_d', G_d', B_d') = \begin{cases} (R_d, G_d, B_d) & \text{if } \text{dstPremultiplied} \text{ is VK\_TRUE} \\ (R_d A_d, G_d A_d, B_d A_d) & \text{if } \text{dstPremultiplied} \text{ is VK\_FALSE} \end{cases}$$

Table 38. Additional RGB Blend Operations

Mode	Result
<code>VK_BLEND_OP_PLUS_EXT</code>	$(R, G, B, A) = (R_s' + R_d', G_s' + G_d', B_s' + B_d', A_s + A_d)$
<code>VK_BLEND_OP_PLUS_CLAMPED_EXT</code>	$(R, G, B, A) = (\min(1, R_s' + R_d'), \min(1, G_s' + G_d'), \min(1, B_s' + B_d'), \min(1, A_s + A_d))$
<code>VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT</code>	$(R, G, B, A) = (\min(\min(1, A_s + A_d), R_s' + R_d'), \min(\min(1, A_s + A_d), G_s' + G_d'), \min(\min(1, A_s + A_d), B_s' + B_d'), \min(1, A_s + A_d))$
<code>VK_BLEND_OP_PLUS_DARKER_EXT</code>	$(R, G, B, A) = (\max(0, \min(1, A_s + A_d) - ((A_s - R_s') + (A_d - R_d'))), \max(0, \min(1, A_s + A_d) - ((A_s - G_s') + (A_d - G_d'))), \max(0, \min(1, A_s + A_d) - ((A_s - B_s') + (A_d - B_d'))), \min(1, A_s + A_d))$
<code>VK_BLEND_OP_MINUS_EXT</code>	$(R, G, B, A) = (R_d' - R_s', G_d' - G_s', B_d' - B_s', A_d - A_s)$

Mode	Result
<code>VK_BLEND_OP_MINUS_CLAMPED_EXT</code>	$(R, G, B, A) = (\max(0, R_d' - R_s'), \max(0, G_d' - G_s'), \max(0, B_d' - B_s'), \max(0, A_d - A_s))$
<code>VK_BLEND_OP_CONTRAST_EXT</code>	$(R, G, B, A) = (\frac{A_d}{2} + 2(R_d' - \frac{A_d}{2})(R_s' - \frac{A_s}{2}), \frac{A_d}{2} + 2(G_d' - \frac{A_d}{2})(G_s' - \frac{A_s}{2}), \frac{A_d}{2} + 2(B_d' - \frac{A_d}{2})(B_s' - \frac{A_s}{2}), A_d)$
<code>VK_BLEND_OP_INVERT_OVG_EXT</code>	$(R, G, B, A) = (A_s(1 - R_d') + (1 - A_s)R_d', A_s(1 - G_d') + (1 - A_s)G_d', A_s(1 - B_d') + (1 - A_s)B_d', A_s + A_d - A_sA_d)$
<code>VK_BLEND_OP_RED_EXT</code>	$(R, G, B, A) = (R_s', G_d', B_d', A_d)$
<code>VK_BLEND_OP_GREEN_EXT</code>	$(R, G, B, A) = (R_d', G_s', B_d', A_d)$
<code>VK_BLEND_OP_BLUE_EXT</code>	$(R, G, B, A) = (R_d', G_d', B_s', A_d)$

## 28.2. Logical Operations

The application can enable a *logical operation* between the fragment's color values and the existing value in the framebuffer attachment. This logical operation is applied prior to updating the framebuffer attachment. Logical operations are applied only for signed and unsigned integer and normalized integer framebuffers. Logical operations are not applied to floating-point or sRGB format color attachments.

Logical operations are controlled by the `logicOpEnable` and `logicOp` members of [VkPipelineColorBlendStateCreateInfo](#). If `logicOpEnable` is `VK_TRUE`, then a logical operation selected by `logicOp` is applied between each color attachment and the fragment's corresponding output value, and blending of all attachments is treated as if it were disabled. Any attachments using color formats for which logical operations are not supported simply pass through the color values unmodified. The logical operation is applied independently for each of the red, green, blue, and alpha components. The `logicOp` is selected from the following operations:

```
typedef enum VkLogicOp {  
    VK_LOGIC_OP_CLEAR = 0,  
    VK_LOGIC_OP_AND = 1,  
    VK_LOGIC_OP_AND_REVERSE = 2,  
    VK_LOGIC_OP_COPY = 3,  
    VK_LOGIC_OP_AND_INVERTED = 4,  
    VK_LOGIC_OP_NO_OP = 5,  
    VK_LOGIC_OP_XOR = 6,  
    VK_LOGIC_OP_OR = 7,  
    VK_LOGIC_OP_NOR = 8,  
    VK_LOGIC_OP_EQUIVALENT = 9,  
    VK_LOGIC_OP_INVERT = 10,  
    VK_LOGIC_OP_OR_REVERSE = 11,  
    VK_LOGIC_OP_COPY_INVERTED = 12,  
    VK_LOGIC_OP_OR_INVERTED = 13,  
    VK_LOGIC_OP_NAND = 14,  
    VK_LOGIC_OP_SET = 15,  
    VK_LOGIC_OP_MAX_ENUM = 0x7FFFFFFF  
} VkLogicOp;
```

The logical operations supported by Vulkan are summarized in the following table in which

- $\neg$  is bitwise invert,
- $\wedge$  is bitwise and,
- $\vee$  is bitwise or,
- $\oplus$  is bitwise exclusive or,
- $s$  is the fragment's  $R_{s0}$ ,  $G_{s0}$ ,  $B_{s0}$  or  $A_{s0}$  component value for the fragment output corresponding to the color attachment being updated, and
- $d$  is the color attachment's R, G, B or A component value:

*Table 39. Logical Operations*

Mode	Operation
<code>VK_LOGIC_OP_CLEAR</code>	$0$
<code>VK_LOGIC_OP_AND</code>	$s \wedge d$
<code>VK_LOGIC_OP_AND_REVERSE</code>	$s \wedge \neg d$
<code>VK_LOGIC_OP_COPY</code>	$s$
<code>VK_LOGIC_OP_AND_INVERTED</code>	$\neg s \wedge d$
<code>VK_LOGIC_OP_NO_OP</code>	$d$
<code>VK_LOGIC_OP_XOR</code>	$s \oplus d$
<code>VK_LOGIC_OP_OR</code>	$s \vee d$
<code>VK_LOGIC_OP_NOR</code>	$\neg(s \vee d)$
<code>VK_LOGIC_OP_EQUIVALENT</code>	$\neg(s \oplus d)$
<code>VK_LOGIC_OP_INVERT</code>	$\neg d$
<code>VK_LOGIC_OP_OR_REVERSE</code>	$s \vee \neg d$
<code>VK_LOGIC_OP_COPY_INVERTED</code>	$\neg s$
<code>VK_LOGIC_OP_OR_INVERTED</code>	$\neg s \vee d$
<code>VK_LOGIC_OP_NAND</code>	$\neg(s \wedge d)$
<code>VK_LOGIC_OP_SET</code>	all 1s

The result of the logical operation is then written to the color attachment as controlled by the component write mask, described in [Blend Operations](#).

## 28.3. Color Write Mask

Bits which **can** be set in `VkPipelineColorBlendAttachmentState::colorWriteMask` to determine whether the final color values R, G, B and A are written to the framebuffer attachment are:

```
typedef enum VkColorComponentFlagBits {
    VK_COLOR_COMPONENT_R_BIT = 0x00000001,
    VK_COLOR_COMPONENT_G_BIT = 0x00000002,
    VK_COLOR_COMPONENT_B_BIT = 0x00000004,
    VK_COLOR_COMPONENT_A_BIT = 0x00000008,
    VK_COLOR_COMPONENT_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkColorComponentFlagBits;
```

- **VK\_COLOR\_COMPONENT\_R\_BIT** specifies that the R value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK\_COLOR\_COMPONENT\_G\_BIT** specifies that the G value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK\_COLOR\_COMPONENT\_B\_BIT** specifies that the B value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK\_COLOR\_COMPONENT\_A\_BIT** specifies that the A value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.

The color write mask operation is applied regardless of whether blending is enabled.

```
typedef VkFlags VkColorComponentFlags;
```

`VkColorComponentFlags` is a bitmask type for setting a mask of zero or more `VkColorComponentFlagBits`.

# Chapter 29. Dispatching Commands

*Dispatching commands* (commands with **Dispatch** in the name) provoke work in a compute pipeline. Dispatching commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound compute pipeline. A compute pipeline **must** be bound to a command buffer before any dispatch commands are recorded in that command buffer.

To record a dispatch, call:

```
void vkCmdDispatch(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    uint32_t  
        commandBuffer,  
        groupCountX,  
        groupCountY,  
        groupCountZ);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **groupCountX** is the number of local workgroups to dispatch in the X dimension.
- **groupCountY** is the number of local workgroups to dispatch in the Y dimension.
- **groupCountZ** is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of **groupCountX** × **groupCountY** × **groupCountZ** local workgroups is assembled.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- If `commandBuffer` is a protected command buffer, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be an unprotected resource
- If `commandBuffer` is a protected command buffer, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point **must** not write to any resource
- `groupCountX` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`
- `groupCountY` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]`
- `groupCountZ` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- This command **must** only be called outside of a render pass instance

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Compute	Compute

To record an indirect command dispatch, call:

```
void vkCmdDispatchIndirect(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
        commandBuffer,  
        buffer,  
        offset);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `buffer` is the buffer containing dispatch parameters.
- `offset` is the byte offset into `buffer` where parameters begin.

`vkCmdDispatchIndirect` behaves similarly to `vkCmdDispatch` except that the parameters are read by the device from a buffer during execution. The parameters of the dispatch are encoded in a `VkDispatchIndirectCommand` structure taken from `buffer` starting at `offset`.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- If `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `offset` **must** be a multiple of 4
- `commandBuffer` **must** not be a protected command buffer
- The sum of `offset` and the size of `VkDispatchIndirectCommand` **must** be less than or equal to the size of `buffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- This command **must** only be called outside of a render pass instance
- Both of `buffer`, and `commandBuffer` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Compute	Compute

The `VkDispatchIndirectCommand` structure is defined as:

```
typedef struct VkDispatchIndirectCommand {
    uint32_t    x;
    uint32_t    y;
    uint32_t    z;
} VkDispatchIndirectCommand;
```

- `x` is the number of local workgroups to dispatch in the X dimension.
- `y` is the number of local workgroups to dispatch in the Y dimension.
- `z` is the number of local workgroups to dispatch in the Z dimension.

The members of `VkDispatchIndirectCommand` have the same meaning as the corresponding parameters of `vkCmdDispatch`.

## Valid Usage

- `x` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`
- `y` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]`
- `z` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]`

To record a dispatch using non-zero base values for the components of `WorkgroupId`, call:

```
void vkCmdDispatchBase(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
        commandBuffer,  
        baseGroupX,  
        baseGroupY,  
        baseGroupZ,  
        groupCountX,  
        groupCountY,  
        groupCountZ);
```

or the equivalent command

```
void vkCmdDispatchBaseKHR(  
    VkCommandBuffer  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
    uint32_t  
        commandBuffer,  
        baseGroupX,  
        baseGroupY,  
        baseGroupZ,  
        groupCountX,  
        groupCountY,  
        groupCountZ);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `baseGroupX` is the start value for the X component of `WorkgroupId`.
- `baseGroupY` is the start value for the Y component of `WorkgroupId`.
- `baseGroupZ` is the start value for the Z component of `WorkgroupId`.
- `groupCountX` is the number of local workgroups to dispatch in the X dimension.
- `groupCountY` is the number of local workgroups to dispatch in the Y dimension.
- `groupCountZ` is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of `groupCountX × groupCountY × groupCountZ` local workgroups is assembled, with `WorkgroupId` values ranging from `[baseGroup*, baseGroup* + groupCount*)` in each component. `vkCmdDispatch` is equivalent to `vkCmdDispatchBase(0, 0, 0, groupCountX, groupCountY, groupCountZ)`.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- `baseGroupX` **must** be less than `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`
- `baseGroupX` **must** be less than `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]`
- `baseGroupZ` **must** be less than `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]`
- `groupCountX` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]` minus `baseGroupX`
- `groupCountY` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]` minus `baseGroupY`
- `groupCountZ` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]` minus `baseGroupZ`
- If any of `baseGroupX`, `baseGroupY`, or `baseGroupZ` are not zero, then the bound compute pipeline **must** have been created with the `VK_PIPELINE_CREATE_DISPATCH_BASE` flag.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- This command **must** only be called outside of a render pass instance

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Compute	

# Chapter 30. Device-Generated Commands

This chapter discusses the generation of command buffer content on the device. These principle steps are to be taken to generate commands on the device:

- Make resource bindings accessible for the device via registering in a `VkObjectTableNVX`.
- Define via `VkIndirectCommandsLayoutNVX` the sequence of commands which should be generated.
- Fill one or more `VkBuffer` with the appropriate content that gets interpreted by `VkIndirectCommandsLayoutNVX`.
- Reserve command space via `vkCmdReserveSpaceForCommandsNVX` in a secondary `VkCommandBuffer` where the generated commands should be recorded.
- Generate the actual commands via `vkCmdProcessCommandsNVX` passing all required data.

Execution of such generated commands can either be triggered directly with the generation process, or by executing the secondary `VkCommandBuffer` that was chosen as optional target. The latter allows re-using generated commands as well. Similar to `VkDescriptorSet`, special care **should** be taken for the lifetime of resources referenced in `VkObjectTableNVX`, which may be accessed at either generation or execution time.

`vkCmdProcessCommandsNVX` executes in a separate logical pipeline from either graphics or compute. When generating commands into a secondary command buffer, the command generation **must** be explicitly synchronized against the secondary command buffer's execution. When not using a secondary command buffer, the command generation is automatically synchronized against the command execution.

## 30.1. Features and Limitations

To query the support of related features and limitations, call:

```
void vkGetPhysicalDeviceGeneratedCommandsPropertiesNVX(
    VkPhysicalDevice                      physicalDevice,
    VkDeviceGeneratedCommandsFeaturesNVX* pFeatures,
    VkDeviceGeneratedCommandsLimitsNVX*   pLimits);
```

- `physicalDevice` is the handle to the physical device whose properties will be queried.
- `pFeatures` is a pointer to a `VkDeviceGeneratedCommandsFeaturesNVX` structure in which features are returned.
- `pLimits` is a pointer to a `VkDeviceGeneratedCommandsLimitsNVX` structure in which limitations are returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pFeatures` **must** be a valid pointer to a `VkDeviceGeneratedCommandsFeaturesNVX` structure
- `pLimits` **must** be a valid pointer to a `VkDeviceGeneratedCommandsLimitsNVX` structure

The `VkDeviceGeneratedCommandsFeaturesNVX` structure is defined as:

```
typedef struct VkDeviceGeneratedCommandsFeaturesNVX {  
    VkStructureType    sType;  
    const void*        pNext;  
    VkBool32           computeBindingPointSupport;  
} VkDeviceGeneratedCommandsFeaturesNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `computeBindingPointSupport` specifies whether the `VkObjectTableNVX` supports entries with `VK_OBJECT_ENTRY_USAGE_GRAPHICS_BIT_NVX` bit set and `VkIndirectCommandsLayoutNVX` supports `VK_PIPELINE_BIND_POINT_COMPUTE`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GENERATED_COMMANDS_FEATURES_NVX`
- `pNext` **must** be `NULL`

The `VkDeviceGeneratedCommandsLimitsNVX` structure is defined as:

```
typedef struct VkDeviceGeneratedCommandsLimitsNVX {  
    VkStructureType    sType;  
    const void*        pNext;  
    uint32_t           maxIndirectCommandsLayoutTokenCount;  
    uint32_t           maxObjectEntryCounts;  
    uint32_t           minSequenceCountBufferOffsetAlignment;  
    uint32_t           minSequenceIndexBufferOffsetAlignment;  
    uint32_t           minCommandsTokenBufferOffsetAlignment;  
} VkDeviceGeneratedCommandsLimitsNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxIndirectCommandsLayoutTokenCount` the maximum number of tokens in `VkIndirectCommandsLayoutNVX`.
- `maxObjectEntryCounts` the maximum number of entries per resource type in `VkObjectTableNVX`.

- `minSequenceCountBufferOffsetAlignment` the minimum alignment for memory addresses optionally used in `vkCmdProcessCommandsNVX`.
- `minSequenceIndexBufferOffsetAlignment` the minimum alignment for memory addresses optionally used in `vkCmdProcessCommandsNVX`.
- `minCommandsTokenBufferOffsetAlignment` the minimum alignment for memory addresses optionally used in `vkCmdProcessCommandsNVX`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GENERATED_COMMANDS_LIMITS_NVX`
- `pNext` **must** be `NULL`

## 30.2. Binding Object Table

The device-side bindings are registered inside a table:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkObjectTableNVX)
```

This is required as the CPU-side object pointers, for example when binding a `VkPipeline` or `VkDescriptorSet`, cannot be used by the device. The combination of `VkObjectTableNVX` and `uint32_t` table indices stored inside a `VkBuffer` serve that purpose during device command generation.

At creation time the table is defined with a fixed amount of registration slots for the individual resource types. A detailed resource binding can then later be registered via `vkRegisterObjectsNVX` at any `uint32_t` index below the allocated maximum.

### 30.2.1. Table Creation

To create object tables, call:

```
VkResult vkCreateObjectTableNVX(
    VkDevice                                     device,
    const VkObjectTableCreateInfoNVX*            pCreateInfo,
    const VkAllocationCallbacks*                 pAllocator,
    VkObjectTableNVX*                           pObjectTable);
```

- `device` is the logical device that creates the object table.
- `pCreateInfo` is a pointer to a `VkObjectTableCreateInfoNVX` structure containing parameters affecting creation of the table.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pObjectTable` is a pointer to a `VkObjectTableNVX` handle in which the resulting object table is returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkObjectTableCreateInfoNVX` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pObjectTable` **must** be a valid pointer to a `VkObjectTableNVX` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkObjectTableCreateInfoNVX` structure is defined as:

```
typedef struct VkObjectTableCreateInfoNVX {
    VkStructureType                     sType;
    const void*                         pNext;
    uint32_t                            objectCount;
    const VkObjectEntryTypeNVX*         pObjectEntryTypes;
    const uint32_t*                     pObjectEntryCounts;
    const VkObjectEntryUsageFlagsNVX*   pObjectEntryUsageFlags;
    uint32_t                            maxUniformBuffersPerDescriptor;
    uint32_t                            maxStorageBuffersPerDescriptor;
    uint32_t                            maxStorageImagesPerDescriptor;
    uint32_t                            maxSampledImagesPerDescriptor;
    uint32_t                            maxPipelineLayouts;
} VkObjectTableCreateInfoNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `objectCount` is the number of entry configurations that the object table supports.
- `pObjectEntryTypes` is a pointer to an array of `VkObjectEntryTypeNVX` values providing the entry type of a given configuration.
- `pObjectEntryCounts` is a pointer to an array of counts of how many objects can be registered in the table.
- `pObjectEntryUsageFlags` is a pointer to an array of bitmasks of `VkObjectEntryUsageFlagBitsNVX` specifying the binding usage of the entry.

- `maxUniformBuffersPerDescriptor` is the maximum number of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` used by any single registered `VkDescriptorSet` in this table.
- `maxStorageBuffersPerDescriptor` is the maximum number of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` used by any single registered `VkDescriptorSet` in this table.
- `maxStorageImagesPerDescriptor` is the maximum number of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` used by any single registered `VkDescriptorSet` in this table.
- `maxSampledImagesPerDescriptor` is the maximum number of `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` used by any single registered `VkDescriptorSet` in this table.
- `maxPipelineLayouts` is the maximum number of unique `VkPipelineLayout` used by any registered `VkDescriptorSet` or `VkPipeline` in this table.

## Valid Usage

- If the `VkDeviceGeneratedCommandsFeaturesNVX::computeBindingPointSupport` feature is not enabled, `pObjectEntryUsageFlags` **must** not contain `VK_OBJECT_ENTRY_USAGE COMPUTE_BIT_NVX`
- Any value within `pObjectEntryCounts` **must** not exceed `VkDeviceGeneratedCommandsLimitsNVX::maxObjectEntryCounts`
- `maxUniformBuffersPerDescriptor` **must** be within the limits supported by the device.
- `maxStorageBuffersPerDescriptor` **must** be within the limits supported by the device.
- `maxStorageImagesPerDescriptor` **must** be within the limits supported by the device.
- `maxSampledImagesPerDescriptor` **must** be within the limits supported by the device.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_OBJECT_TABLE_CREATE_INFO_NVX`
- `pNext` **must** be `NULL`
- `pObjectEntryTypes` **must** be a valid pointer to an array of `objectCount` valid `VkObjectEntryTypeNVX` values
- `pObjectEntryCounts` **must** be a valid pointer to an array of `objectCount uint32_t` values
- `pObjectEntryUsageFlags` **must** be a valid pointer to an array of `objectCount` valid combinations of `VkObjectEntryUsageFlagBitsNVX` values
- Each element of `pObjectEntryUsageFlags` **must** not be `0`
- `objectCount` **must** be greater than `0`

Possible values of elements of the `VkObjectTableCreateInfoNVX::pObjectEntryTypes` array, specifying the entry type of a configuration, are:

```

typedef enum VkObjectEntryTypeNVX {
    VK_OBJECT_ENTRY_TYPE_DESCRIPTOR_SET_NVX = 0,
    VK_OBJECT_ENTRY_TYPE_PIPELINE_NVX = 1,
    VK_OBJECT_ENTRY_TYPE_INDEX_BUFFER_NVX = 2,
    VK_OBJECT_ENTRY_TYPE_VERTEX_BUFFER_NVX = 3,
    VK_OBJECT_ENTRY_TYPE_PUSH_CONSTANT_NVX = 4,
    VK_OBJECT_ENTRY_TYPE_MAX_ENUM_NVX = 0x7FFFFFFF
} VkObjectEntryTypeNVX;

```

- `VK_OBJECT_ENTRY_TYPE_DESCRIPTOR_SET_NVX` specifies a `VkDescriptorSet` resource entry that is registered via `VkObjectTableDescriptorSetEntryNVX`.
- `VK_OBJECT_ENTRY_TYPE_PIPELINE_NVX` specifies a `VkPipeline` resource entry that is registered via `VkObjectTablePipelineEntryNVX`.
- `VK_OBJECT_ENTRY_TYPE_INDEX_BUFFER_NVX` specifies a `VkBuffer` resource entry that is registered via `VkObjectTableIndexBufferEntryNVX`.
- `VK_OBJECT_ENTRY_TYPE_VERTEX_BUFFER_NVX` specifies a `VkBuffer` resource entry that is registered via `VkObjectTableVertexBufferEntryNVX`.
- `VK_OBJECT_ENTRY_TYPE_PUSH_CONSTANT_NVX` specifies the resource entry is registered via `VkObjectTablePushConstantEntryNVX`.

Bits which **can** be set in elements of the `VkObjectTableCreateInfoNVX::pObjectEntryUsageFlags` array, specifying binding usage of an entry, are:

```

typedef enum VkObjectEntryUsageFlagBitsNVX {
    VK_OBJECT_ENTRY_USAGE_GRAPHICS_BIT_NVX = 0x00000001,
    VK_OBJECT_ENTRY_USAGE_COMPUTE_BIT_NVX = 0x00000002,
    VK_OBJECT_ENTRY_USAGE_FLAG_BITS_MAX_ENUM_NVX = 0x7FFFFFFF
} VkObjectEntryUsageFlagBitsNVX;

```

- `VK_OBJECT_ENTRY_USAGE_GRAPHICS_BIT_NVX` specifies that the resource is bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`
- `VK_OBJECT_ENTRY_USAGE_COMPUTE_BIT_NVX` specifies that the resource is bound to `VK_PIPELINE_BIND_POINT_COMPUTE`

```
typedef VkFlags VkObjectEntryUsageFlagsNVX;
```

`VkObjectEntryUsageFlagsNVX` is a bitmask type for setting a mask of zero or more `VkObjectEntryUsageFlagBitsNVX`.

To destroy an object table, call:

```
void vkDestroyObjectTableNVX(  
    VkDevice  
    VkObjectTableNVX  
    const VkAllocationCallbacks*  
                                device,  
                                objectTable,  
                                pAllocator);
```

- `device` is the logical device that destroys the table.
- `objectTable` is the table to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

### Valid Usage

- All submitted commands that refer to `objectTable` **must** have completed execution.
- If `VkAllocationCallbacks` were provided when `objectTable` was created, a compatible set of callbacks **must** be provided here.
- If no `VkAllocationCallbacks` were provided when `objectTable` was created, `pAllocator` **must** be `NULL`.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `objectTable` **must** be a valid `VkObjectTableNVX` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `objectTable` **must** have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `objectTable` **must** be externally synchronized

## 30.2.2. Registering Objects

Resource bindings of Vulkan objects are registered at an arbitrary `uint32_t` index within an object table. As long as the object table references such objects, they **must** not be deleted.

```
VkResult vkRegisterObjectsNVX(  
    VkDevice  
    VkObjectTableNVX  
    uint32_t  
    const VkObjectTableEntryNVX* const*  
    const uint32_t*  
                                device,  
                                objectTable,  
                                objectCount,  
                                ppObjectTableEntries,  
                                pObjectIndices);
```

- `device` is the logical device that creates the object table.
- `objectTable` is the table for which the resources are registered.
- `objectCount` is the number of resources to register.
- `ppObjectTableEntries` provides an array for detailed binding informations. Each array element is a pointer to a structure of type `VkObjectTablePipelineEntryNVX`, `VkObjectTableDescriptorSetEntryNVX`, `VkObjectTableVertexBufferEntryNVX`, `VkObjectTableIndexBufferEntryNVX` or `VkObjectTablePushConstantEntryNVX` (see below for details).
- `pObjectIndices` are the indices at which each resource is registered.

## Valid Usage

- The contents of `pObjectTableEntry` **must** yield plausible bindings supported by the device.
- At any `pObjectIndices` there **must** not be a registered resource already.
- Any value inside `pObjectIndices` **must** be below the appropriate `VkObjectTableCreateInfoNVX::pObjectEntryCounts` limits provided at `objectTable` creation time.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `objectTable` **must** be a valid `VkObjectTableNVX` handle
- `ppObjectTableEntries` **must** be a valid pointer to an array of `objectCount` valid `VkObjectTableEntryNVX` structures
- `pObjectIndices` **must** be a valid pointer to an array of `objectCount uint32_t` values
- `objectCount` **must** be greater than `0`
- `objectTable` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `objectTable` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Common to all resource entries are:

```
typedef struct VkObjectTableEntryNVX {  
    VkObjectTypeNVX          type;  
    VkObjectEntryUsageFlagsNVX flags;  
} VkObjectTableEntryNVX;
```

- **type** defines the entry type
- **flags** defines which [VkPipelineBindPoint](#) the resource can be used with. Some entry types allow only a single flag to be set.

## Valid Usage

- If the [VkDeviceGeneratedCommandsFeaturesNVX::computeBindingPointSupport](#) feature is not enabled, **flags must** not contain [VK\\_OBJECT\\_ENTRY\\_USAGE\\_COMPUTE\\_BIT\\_NVX](#)

## Valid Usage (Implicit)

- **type must** be a valid [VkObjectTypeNVX](#) value
- **flags must** be a valid combination of [VkObjectEntryUsageFlagBitsNVX](#) values
- **flags must** not be `0`

```
typedef struct VkObjectTablePipelineEntryNVX {  
    VkObjectTypeNVX          type;  
    VkObjectEntryUsageFlagsNVX flags;  
    VkPipeline               pipeline;  
} VkObjectTablePipelineEntryNVX;
```

- **pipeline** specifies the [VkPipeline](#) that this resource entry references.

## Valid Usage

- **type must** be [VK\\_OBJECT\\_ENTRY\\_TYPE\\_PIPELINE\\_NVX](#)

## Valid Usage (Implicit)

- **type must** be a valid [VkObjectTypeNVX](#) value
- **flags must** be a valid combination of [VkObjectEntryUsageFlagBitsNVX](#) values
- **flags must** not be `0`
- **pipeline must** be a valid [VkPipeline](#) handle

```
typedef struct VkObjectTableDescriptorSetEntryNVX {
    VkObjectEntryTypeNVX          type;
    VkObjectEntryUsageFlagsNVX    flags;
    VkPipelineLayout               pipelineLayout;
    VkDescriptorSet                descriptorSet;
} VkObjectTableDescriptorSetEntryNVX;
```

- `pipelineLayout` specifies the `VkPipelineLayout` that the `descriptorSet` is used with.
- `descriptorSet` specifies the `VkDescriptorSet` that can be bound with this entry.

### Valid Usage

- `type` **must** be `VK_OBJECT_ENTRY_TYPE_DESCRIPTOR_SET_NVX`

### Valid Usage (Implicit)

- `type` **must** be a valid `VkObjectEntryTypeNVX` value
- `flags` **must** be a valid combination of `VkObjectEntryUsageFlagBitsNVX` values
- `flags` **must** not be `0`
- `pipelineLayout` **must** be a valid `VkPipelineLayout` handle
- `descriptorSet` **must** be a valid `VkDescriptorSet` handle
- Both of `descriptorSet`, and `pipelineLayout` **must** have been created, allocated, or retrieved from the same `VkDevice`

```
typedef struct VkObjectTableVertexBufferEntryNVX {
    VkObjectEntryTypeNVX          type;
    VkObjectEntryUsageFlagsNVX    flags;
    VkBuffer                      buffer;
} VkObjectTableVertexBufferEntryNVX;
```

- `buffer` specifies the `VkBuffer` that can be bound as vertex bufer

### Valid Usage

- `type` **must** be `VK_OBJECT_ENTRY_TYPE_VERTEX_BUFFER_NVX`

## Valid Usage (Implicit)

- **type** **must** be a valid [VkObjectTypeNVX](#) value
- **flags** **must** be a valid combination of [VkObjectEntryUsageFlagBitsNVX](#) values
- **flags** **must** not be `0`
- **buffer** **must** be a valid [VkBuffer](#) handle

```
typedef struct VkObjectTableIndexBufferEntryNVX {  
    VkObjectTypeNVX          type;  
    VkObjectEntryUsageFlagsNVX flags;  
    VkBuffer                 buffer;  
    VkIndexType               indexType;  
} VkObjectTableIndexBufferEntryNVX;
```

- **buffer** specifies the [VkBuffer](#) that can be bound as index buffer
- **indexType** specifies the [VkIndexType](#) used with this index buffer

## Valid Usage

- **type** **must** be `VK_OBJECT_ENTRY_TYPE_INDEX_BUFFER_NVX`
- **indexType** **must** be `VK_INDEX_TYPE_UINT16`, or `VK_INDEX_TYPE_UINT32`

## Valid Usage (Implicit)

- **type** **must** be a valid [VkObjectTypeNVX](#) value
- **flags** **must** be a valid combination of [VkObjectEntryUsageFlagBitsNVX](#) values
- **flags** **must** not be `0`
- **buffer** **must** be a valid [VkBuffer](#) handle
- **indexType** **must** be a valid [VkIndexType](#) value

```
typedef struct VkObjectTablePushConstantEntryNVX {  
    VkObjectTypeNVX          type;  
    VkObjectEntryUsageFlagsNVX flags;  
    VkPipelineLayout          pipelineLayout;  
    VkShaderStageFlags        stageFlags;  
} VkObjectTablePushConstantEntryNVX;
```

- **pipelineLayout** specifies the [VkPipelineLayout](#) that the pushconstants are used with
- **stageFlags** specifies the [VkShaderStageFlags](#) that the pushconstants are used with

## Valid Usage

- **type** **must** be `VK_OBJECT_ENTRY_TYPE_PUSH_CONSTANT_NVX`

## Valid Usage (Implicit)

- **type** **must** be a valid `VkObjectEntryTypeNVX` value
- **flags** **must** be a valid combination of `VkObjectEntryUsageFlagBitsNVX` values
- **flags** **must** not be `0`
- **pipelineLayout** **must** be a valid `VkPipelineLayout` handle
- **stageFlags** **must** be a valid combination of `VkShaderStageFlagBits` values
- **stageFlags** **must** not be `0`

Use the following command to unregister resources from an object table:

```
VkResult vkUnregisterObjectsNVX(  
    VkDevice                                     device,  
    VkObjectTableNVX                            objectTable,  
    uint32_t                                     objectCount,  
    const VkObjectEntryTypeNVX*                  pObjectEntryTypes,  
    const uint32_t*                             pObjectIndices);
```

- **device** is the logical device that creates the object table.
- **objectTable** is the table from which the resources are unregistered.
- **objectCount** is the number of resources being removed from the object table.
- **pObjectEntryType** provides an array of `VkObjectEntryTypeNVX` for the resources being removed.
- **pObjectIndices** provides the array of object indices to be removed.

## Valid Usage

- At any **pObjectIndices** there **must** be a registered resource already.
- The **pObjectEntryTypes** of the resource at **pObjectIndices** **must** match.
- All operations on the device using the registered resource **must** have been completed.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `objectTable` **must** be a valid `VkObjectTableNVX` handle
- `pObjectEntryTypes` **must** be a valid pointer to an array of `objectCount` valid `VkObjectEntryTypeNVX` values
- `pObjectIndices` **must** be a valid pointer to an array of `objectCount uint32_t` values
- `objectCount` **must** be greater than 0
- `objectTable` **must** have been created, allocated, or retrieved from `device`

## Host Synchronization

- Host access to `objectTable` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

## 30.3. Indirect Commands Layout

The device-side command generation happens through an iterative processing of an atomic sequence comprised of command tokens, which are represented by:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkIndirectCommandsLayoutNVX)
```

### 30.3.1. Tokenized Command Processing

The processing is in principle illustrated below:

```

void cmdProcessSequence(cmd, objectTable, indirectCommandsLayout,
pIndirectCommandsTokens, s)
{
    for (c = 0; c < indirectCommandsLayout.tokenCount; c++)
    {
        indirectCommandsLayout.pTokens[c].command (cmd, objectTable,
pIndirectCommandsTokens[c], s);
    }
}

void cmdProcessAllSequences(cmd, objectTable, indirectCommandsLayout,
pIndirectCommandsTokens, sequencesCount)
{
    for (s = 0; s < sequencesCount; s++)
    {
        cmdProcessSequence(cmd, objectTable, indirectCommandsLayout,
pIndirectCommandsTokens, s);
    }
}

```

The processing of each sequence is considered stateless, therefore all state changes **must** occur prior work provoking commands within the sequence. A single sequence is either strictly targeting [VK\\_PIPELINE\\_BIND\\_POINT\\_GRAPHICS](#) or [VK\\_PIPELINE\\_BIND\\_POINT\\_COMPUTE](#).

The primary input data for each token is provided through [VkBuffer](#) content at command generation time using [vkCmdProcessCommandsNVX](#), however some functional arguments, for example binding sets, are specified at layout creation time. The input size is different for each token.

Possible values of those elements of the [VkIndirectCommandsLayoutCreateInfoNVX::pTokens](#) array which specify command tokens (other elements of the array specify command parameters) are:

```

typedef enum VkIndirectCommandsTokenTypeNVX {
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_PIPELINE_NVX = 0,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DESCRIPTOR_SET_NVX = 1,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_INDEX_BUFFER_NVX = 2,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_VERTEX_BUFFER_NVX = 3,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NVX = 4,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_INDEXED_NVX = 5,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_NVX = 6,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DISPATCH_NVX = 7,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_MAX_ENUM_NVX = 0x7FFFFFFF
} VkIndirectCommandsTokenTypeNVX;

```

*Table 40. Supported indirect command tokens*

Token type	Equivalent command
<a href="#">VK_INDIRECT_COMMANDS_TOKEN_TYPE_PIPELINE_NVX</a>	<a href="#">vkCmdBindPipeline</a>

Token type	Equivalent command
VK INDIRECT COMMANDS TOKEN TYPE DESCRIPTOR SET NVX	vkCmdBindDescriptorSets
VK INDIRECT COMMANDS TOKEN TYPE INDEX BUFFER NVX	vkCmdBindIndexBuffer
VK INDIRECT COMMANDS TOKEN TYPE VERTEX BUFFER NVX	vkCmdBindVertexBuffers
VK INDIRECT COMMANDS TOKEN TYPE PUSH CONSTANT NVX	vkCmdPushConstants
VK INDIRECT COMMANDS TOKEN TYPE DRAW INDEXED NVX	vkCmdDrawIndexedIndirect
VK INDIRECT COMMANDS TOKEN TYPE DRAW NVX	vkCmdDrawIndirect
VK INDIRECT COMMANDS TOKEN TYPE DISPATCH NVX	vkCmdDispatchIndirect

The `VkIndirectCommandsLayoutTokenNVX` structure specifies details to the function arguments that need to be known at layout creation time:

```
typedef struct VkIndirectCommandsLayoutTokenNVX {
    VkIndirectCommandsTokenTypeNVX    tokenType;
    uint32_t                         bindingUnit;
    uint32_t                         dynamicCount;
    uint32_t                         divisor;
} VkIndirectCommandsLayoutTokenNVX;
```

- `type` specifies the token command type.
- `bindingUnit` has a different meaning depending on the type, please refer pseudo code further down for details.
- `dynamicCount` has a different meaning depending on the type, please refer pseudo code further down for details.
- `divisor` defines the rate at which the input data buffers are accessed.

### Valid Usage

- `bindingUnit` **must** stay within device supported limits for the appropriate commands.
- `dynamicCount` **must** stay within device supported limits for the appropriate commands.
- `divisor` **must** be greater than `0` and a power of two.

### Valid Usage (Implicit)

- `tokenType` **must** be a valid `VkIndirectCommandsTokenTypeNVX` value

The `VkIndirectCommandsTokenNVX` structure specifies the input data for a token at processing time.

```

typedef struct VkIndirectCommandsTokenNVX {
    VkIndirectCommandsTokenTypeNVX    tokenType;
    VkBuffer                         buffer;
    VkDeviceSize                      offset;
} VkIndirectCommandsTokenNVX;

```

- **tokenType** specifies the token command type.
- **buffer** specifies the **VkBuffer** storing the functional arguments for each sequence. These arguments can be written by the device.
- **offset** specifies an offset into **buffer** where the arguments start.

## Valid Usage

- The **buffer**'s usage flag **must** have the **VK\_BUFFER\_USAGE\_INDIRECT\_BUFFER\_BIT** bit set.
- The **offset** **must** be aligned to **VkDeviceGeneratedCommandsLimitsNVX::minCommandsTokenBufferOffsetAlignment**.

## Valid Usage (Implicit)

- **tokenType** **must** be a valid **VkIndirectCommandsTokenTypeNVX** value
- **buffer** **must** be a valid **VkBuffer** handle

The following code provides detailed information on how an individual sequence is processed:

```

void cmdProcessSequence(cmd, objectTable, indirectCommandsLayout,
pIndirectCommandsTokens, s)
{
    for (uint32_t c = 0; c < indirectCommandsLayout.tokenCount; c++){
        input    = pIndirectCommandsTokens[c];
        i       = s / indirectCommandsLayout.pTokens[c].divisor;

        switch(input.type){
            VK_INDIRECT_COMMANDS_TOKEN_TYPE_PIPELINE_NVX:
                size_t    stride  = sizeof(uint32_t);
                uint32_t* data    = input.buffer.pointer( input.offset + stride * i );
                uint32_t object  = data[0];

                vkCmdBindPipeline(cmd, indirectCommandsLayout.pipelineBindPoint,
                                  objectTable.pipelines[ object ].pipeline);
                break;

            VK_INDIRECT_COMMANDS_TOKEN_TYPE_DESCRIPTOR_SET_NVX:
                size_t    stride  = sizeof(uint32_t) + sizeof(uint32_t) *
indirectCommandsLayout.pTokens[c].dynamicCount;

```

```

    uint32_t* data    = input.buffer.pointer( input.offset + stride * i );
    uint32_t object  = data[0];

    vkCmdBindDescriptorSets(cmd, indirectCommandsLayout.pipelineBindPoint,
        objectTable.descriptorsets[ object ].layout,
        indirectCommandsLayout.pTokens[ c ].bindingUnit,
        1, &objectTable.descriptorsets[ object ].descriptorSet,
        indirectCommandsLayout.pTokens[ c ].dynamicCount, data + 1);
    break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NVX:
    size_t    stride  = sizeof(uint32_t) + indirectCommandsLayout.pTokens[c]
.dYNAMICCOUNT;
    uint32_t* data    = input.buffer.pointer( input.offset + stride * i );
    uint32_t object  = data[0];

    vkCmdPushConstants(
        objectTable.pushconstants[ object ].layout,
        objectTable.pushconstants[ object ].stageFlags,
        indirectCommandsLayout.pTokens[ c ].bindingUnit, indirectCommandsLayout
.pTokens[c].dynamicCount, data + 1);
    break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_INDEX_BUFFER_NVX:
    size_t    stride  = sizeof(uint32_t) + sizeof(uint32_t) *
indirectCommandsLayout.pTokens[c].dynamicCount;
    uint32_t* data    = input.buffer.pointer( input.offset + stride * i );
    uint32_t object  = data[0];

    vkCmdBindIndexBuffer(cmd,
        objectTable.vertexbuffers[ object ].buffer,
        indirectCommandsLayout.pTokens[ c ].dynamicCount ? data[1] : 0,
        objectTable.vertexbuffers[ object ].indexType);
    break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_VERTEX_BUFFER_NVX:
    size_t    stride  = sizeof(uint32_t) + sizeof(uint32_t) *
indirectCommandsLayout.pTokens[c].dynamicCount;
    uint32_t* data    = input.buffer.pointer( input.offset + stride * i );
    uint32_t object  = data[0];

    vkCmdBindVertexBuffers(cmd,
        indirectCommandsLayout.pTokens[ c ].bindingUnit, 1,
        &objectTable.vertexbuffers[ object ].buffer,
        indirectCommandsLayout.pTokens[ c ].dynamicCount ? data + 1 : {0}); // device size handled as uint32_t
    break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_INDEXED_NVX:
    vkCmdDrawIndexedIndirect(cmd,
        input.buffer,

```

```

        sizeof(VkDrawIndexedIndirectCommand) * i + input.offset, 1, 0);
    break;

    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_NVX:
        vkCmdDrawIndirect(cmd,
            input.buffer,
            sizeof(VkDrawIndirectCommand) * i + input.offset, 1, 0);
    break;

    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DISPATCH_NVX:
        vkCmdDispatchIndirect(cmd,
            input.buffer,
            sizeof(VkDispatchIndirectCommand) * i + input.offset);
    break;
}
}
}

```

### 30.3.2. Creation and Deletion

Indirect command layouts are created by:

```

VkResult vkCreateIndirectCommandsLayoutNVX(
    VkDevice                      device,
    const VkIndirectCommandsLayoutCreateInfoNVX* pCreateInfo,
    const VkAllocationCallbacks*      pAllocator,
    VkIndirectCommandsLayoutNVX*     pIndirectCommandsLayout);

```

- `device` is the logical device that creates the indirect command layout.
- `pCreateInfo` is a pointer to a `VkIndirectCommandsLayoutCreateInfoNVX` structure containing parameters affecting creation of the indirect command layout.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pIndirectCommandsLayout` is a pointer to a `VkIndirectCommandsLayoutNVX` handle in which the resulting indirect command layout is returned.

#### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkIndirectCommandsLayoutCreateInfoNVX` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pIndirectCommandsLayout` **must** be a valid pointer to a `VkIndirectCommandsLayoutNVX` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkIndirectCommandsLayoutCreateInfoNVX` structure is defined as:

```
typedef struct VkIndirectCommandsLayoutCreateInfoNVX {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineBindPoint                 pipelineBindPoint;
    VkIndirectCommandsLayoutUsageFlagsNVX flags;
    uint32_t                            tokenCount;
    const VkIndirectCommandsLayoutTokenNVX* pTokens;
} VkIndirectCommandsLayoutCreateInfoNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pipelineBindPoint` is the `VkPipelineBindPoint` that this layout targets.
- `flags` is a bitmask of `VkIndirectCommandsLayoutUsageFlagBitsNVX` specifying usage hints of this layout.
- `tokenCount` is the length of the individual command sequence.
- `pTokens` is an array describing each command token in detail. See `VkIndirectCommandsTokenTypeNVX` and `VkIndirectCommandsLayoutTokenNVX` below for details.

The following code illustrates some of the key flags:

```

void cmdProcessAllSequences(cmd, objectTable, indirectCommandsLayout,
pIndirectCommandsTokens, sequencesCount, indexbuffer, indexbufferoffset)
{
    for (s = 0; s < sequencesCount; s++)
    {
        sequence = s;

        if (indirectCommandsLayout.flags &
VK_INDIRECT_COMMANDS_LAYOUT_USAGE_UNORDERED_SEQUENCES_BIT_NVX) {
            sequence = incoherent_implementation_dependent_permutation[ sequence ];
        }
        if (indirectCommandsLayout.flags &
VK_INDIRECT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NVX) {
            sequence = indexbuffer.load_uint32( sequence * sizeof(uint32_t) +
indexbufferoffset);
        }

        cmdProcessSequence( cmd, objectTable, indirectCommandsLayout,
pIndirectCommandsTokens, sequence );
    }
}

```

## Valid Usage

- `tokenCount` **must** be greater than `0` and below `VkDeviceGeneratedCommandsLimitsNVX::maxIndirectCommandsLayoutTokenCount`
- If the `VkDeviceGeneratedCommandsFeaturesNVX::computeBindingPointSupport` feature is not enabled, then `pipelineBindPoint` **must** not be `VK_PIPELINE_BIND_POINT_COMPUTE`
- If `pTokens` contains an entry of `VK_INDIRECT_COMMANDS_TOKEN_TYPE_PIPELINE_NVX` it **must** be the first element of the array and there **must** be only a single element of such token type.
- All state binding tokens in `pTokens` **must** occur prior work provoking tokens (`VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_NVX`,  
`VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_INDEXED_NVX`,  
`VK_INDIRECT_COMMANDS_TOKEN_TYPE_DISPATCH_NVX`).
- The content of `pTokens` **must** include one single work provoking token that is compatible with the `pipelineBindPoint`.

## Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_CREATE_INFO_NVX`
- **pNext** **must** be `NULL`
- **pipelineBindPoint** **must** be a valid `VkPipelineBindPoint` value
- **flags** **must** be a valid combination of `VkIndirectCommandsLayoutUsageFlagBitsNVX` values
- **flags** **must** not be `0`
- **pTokens** **must** be a valid pointer to an array of **tokenCount** valid `VkIndirectCommandsLayoutTokenNVX` structures
- **tokenCount** **must** be greater than `0`

Bits which **can** be set in `VkIndirectCommandsLayoutCreateInfoNVX::flags`, specifying usage hints of an indirect command layout, are:

```
typedef enum VkIndirectCommandsLayoutUsageFlagBitsNVX {
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_UNORDERED_SEQUENCES_BIT_NVX = 0x00000001,
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_SPARSE_SEQUENCES_BIT_NVX = 0x00000002,
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_EMPTY_EXECUTIONS_BIT_NVX = 0x00000004,
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NVX = 0x00000008,
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_FLAG_BITS_MAX_ENUM_NVX = 0x7FFFFFFF
} VkIndirectCommandsLayoutUsageFlagBitsNVX;
```

- `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_UNORDERED_SEQUENCES_BIT_NVX` specifies that the processing of sequences **can** happen at an implementation-dependent order, which is not guaranteed to be coherent across multiple invocations.
- `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_SPARSE_SEQUENCES_BIT_NVX` specifies that there is likely a high difference between allocated number of sequences and actually used.
- `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_EMPTY_EXECUTIONS_BIT_NVX` specifies that there are likely many draw or dispatch calls that are zero-sized (zero grid dimension, no primitives to render).
- `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NVX` specifies that the input data for the sequences is not implicitly indexed from `0..sequencesUsed` but a user provided `VkBuffer` encoding the index is provided.

```
typedef VkFlags VkIndirectCommandsLayoutUsageFlagsNVX;
```

`VkIndirectCommandsLayoutUsageFlagsNVX` is a bitmask type for setting a mask of zero or more `VkIndirectCommandsLayoutUsageFlagBitsNVX`.

Indirect command layouts are destroyed by:

```
void vkDestroyIndirectCommandsLayoutNVX(  
    VkDevice device,  
    VkIndirectCommandsLayoutNVX indirectCommandsLayout,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the layout.
- `indirectCommandsLayout` is the table to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

### Valid Usage

- All submitted commands that refer to `indirectCommandsLayout` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `objectTable` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `objectTable` was created, `pAllocator` **must** be `NULL`

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `indirectCommandsLayout` **must** be a valid `VkIndirectCommandsLayoutNVX` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `indirectCommandsLayout` **must** have been created, allocated, or retrieved from `device`

## 30.4. Indirect Commands Generation

Command space for generated commands recorded into a secondary command buffer **must** be reserved by calling:

```
void vkCmdReserveSpaceForCommandsNVX(  
    VkCommandBuffer commandBuffer,  
    const VkCmdReserveSpaceForCommandsInfoNVX* pReserveSpaceInfo);
```

- `commandBuffer` is the secondary command buffer in which the space for device-generated commands is reserved.
- `pProcessCommandsInfo` is a pointer to a `VkCmdReserveSpaceForCommandsInfoNVX` structure containing parameters affecting the reservation of command buffer space.

## Valid Usage

- The provided `commandBuffer` **must** not have had a prior space reservation since its creation or the last reset.
- The state of the `commandBuffer` **must** be legal to execute all commands within the sequence provided by the `indirectCommandsLayout` member of `pProcessCommandsInfo`.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pReserveSpaceInfo` **must** be a valid pointer to a valid `VkCmdReserveSpaceForCommandsInfoNVX` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- This command **must** only be called inside of a render pass instance
- `commandBuffer` **must** be a secondary `VkCommandBuffer`

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Secondary	Inside	Graphics Compute	

```
typedef struct VkCmdReserveSpaceForCommandsInfoNVX {
    VkStructureType           sType;
    const void*                pNext;
    VkObjectTableNVX          objectTable;
    VkIndirectCommandsLayoutNVX indirectCommandsLayout;
    uint32_t                  maxSequencesCount;
} VkCmdReserveSpaceForCommandsInfoNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `objectTable` is the `VkObjectTableNVX` to be used for the generation process. Only registered objects at the time `vkCmdReserveSpaceForCommandsNVX` is called, will be taken into account for the reservation.
- `indirectCommandsLayout` is the `VkIndirectCommandsLayoutNVX` that **must** also be used at generation time.
- `maxSequencesCount` is the maximum number of sequences for which command buffer space will be reserved.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_CMD_RESERVE_SPACE_FOR_COMMANDS_INFO_NVX`
- `pNext` **must** be `NULL`
- `objectTable` **must** be a valid `VkObjectTableNVX` handle
- `indirectCommandsLayout` **must** be a valid `VkIndirectCommandsLayoutNVX` handle
- Both of `indirectCommandsLayout`, and `objectTable` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `objectTable` **must** be externally synchronized

The generated commands will behave as if they were recorded within the call to `vkCmdReserveSpaceForCommandsNVX`, that means they can inherit state defined in the command buffer prior this call. However, given the stateless nature of the generated sequences, they will not affect commands after the reserved space. Treat the state that **can** be affected by the provided `VkIndirectCommandsLayoutNVX` as undefined.

The actual generation on the device is handled with:

```
void vkCmdProcessCommandsNVX(
    VkCommandBuffer                                commandBuffer,
    const VkCmdProcessCommandsInfoNVX* pProcessCommandsInfo);
```

- `commandBuffer` is the primary command buffer in which the generation process takes space.
- `pProcessCommandsInfo` is a pointer to a `VkCmdProcessCommandsInfoNVX` structure containing parameters affecting the processing of commands.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pProcessCommandsInfo` **must** be a valid pointer to a valid `VkCmdProcessCommandsInfoNVX` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- This command **must** only be called inside of a render pass instance

## Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Inside	Graphics Compute	

```
typedef struct VkCmdProcessCommandsInfoNVX {
    VkStructureType           sType;
    const void*                pNext;
    VkObjectTableNVX          objectTable;
    VkIndirectCommandsLayoutNVX indirectCommandsLayout;
    uint32_t                  indirectCommandsTokenCount;
    const VkIndirectCommandsTokenNVX* pIndirectCommandsTokens;
    uint32_t                  maxSequencesCount;
    VkCommandBuffer            targetCommandBuffer;
    VkBuffer                   sequencesCountBuffer;
    VkDeviceSize               sequencesCountOffset;
    VkBuffer                   sequencesIndexBuffer;
    VkDeviceSize               sequencesIndexOffset;
} VkCmdProcessCommandsInfoNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `objectTable` is the `VkObjectTableNVX` to be used for the generation process. Only registered

objects at the time `vkCmdReserveSpaceForCommandsNVX` is called, will be taken into account for the reservation.

- `indirectCommandsLayout` is the `VkIndirectCommandsLayoutNVX` that provides the command sequence to generate.
- `indirectCommandsTokenCount` defines the number of input tokens used.
- `pIndirectCommandsTokens` provides an array of `VkIndirectCommandsTokenNVX` that reference the input data for each token command.
- `maxSequencesCount` is the maximum number of sequences for which command buffer space will be reserved. If `sequencesCountBuffer` is `VK_NULL_HANDLE`, this is also the actual number of sequences generated.
- `targetCommandBuffer` **can** be the secondary `VkCommandBuffer` in which the commands should be recorded. If `targetCommandBuffer` is `NULL` an implicit reservation as well as execution takes place on the processing `VkCommandBuffer`.
- `sequencesCountBuffer` **can** be `VkBuffer` from which the actual amount of sequences is sourced from as `uint32_t` value.
- `sequencesCountOffset` is the byte offset into `sequencesCountBuffer` where the count value is stored.
- `sequencesIndexBuffer` **must** be set if `indirectCommandsLayout`'s `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NVX` is set and provides the used sequence indices as `uint32_t` array. Otherwise it **must** be `VK_NULL_HANDLE`.
- `sequencesIndexOffset` is the byte offset into `sequencesIndexBuffer` where the index values start.

## Valid Usage

- The provided `objectTable` **must** include all objects referenced by the generation process
- `indirectCommandsTokenCount` **must** match the `indirectCommandsLayout`'s `tokenCount`
- The `tokenType` member of each entry in the `pIndirectCommandsTokens` array **must** match the values used at creation time of `indirectCommandsLayout`
- If `targetCommandBuffer` is provided, it **must** have reserved command space
- If `targetCommandBuffer` is provided, the `objectTable` **must** match the reservation's `objectTable` and **must** have had all referenced objects registered at reservation time
- If `targetCommandBuffer` is provided, the `indirectCommandsLayout` **must** match the reservation's `indirectCommandsLayout`
- If `targetCommandBuffer` is provided, the `maxSequencesCount` **must** not exceed the reservation's `maxSequencesCount`
- If `sequencesCountBuffer` is used, its usage flag **must** have the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- If `sequencesCountBuffer` is used, `sequencesCountOffset` **must** be aligned to `VkDeviceGeneratedCommandsLimitsNVX::minSequenceCountBufferOffsetAlignment`
- If `sequencesIndexBuffer` is used, its usage flag **must** have the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- If `sequencesIndexBuffer` is used, `sequencesIndexOffset` **must** be aligned to `VkDeviceGeneratedCommandsLimitsNVX::minSequenceIndexBufferOffsetAlignment`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_CMD_PROCESS_COMMANDS_INFO_NVX`
- `pNext` **must** be `NULL`
- `objectTable` **must** be a valid `VkObjectTableNVX` handle
- `indirectCommandsLayout` **must** be a valid `VkIndirectCommandsLayoutNVX` handle
- `pIndirectCommandsTokens` **must** be a valid pointer to an array of `indirectCommandsTokenCount` valid `VkIndirectCommandsTokenNVX` structures
- If `targetCommandBuffer` is not `NULL`, `targetCommandBuffer` **must** be a valid `VkCommandBuffer` handle
- If `sequencesCountBuffer` is not `VK_NULL_HANDLE`, `sequencesCountBuffer` **must** be a valid `VkBuffer` handle
- If `sequencesIndexBuffer` is not `VK_NULL_HANDLE`, `sequencesIndexBuffer` **must** be a valid `VkBuffer` handle
- `indirectCommandsTokenCount` **must** be greater than `0`
- Each of `indirectCommandsLayout`, `objectTable`, `sequencesCountBuffer`, `sequencesIndexBuffer`, and `targetCommandBuffer` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `objectTable` **must** be externally synchronized
- Host access to `targetCommandBuffer` **must** be externally synchronized

Referencing the functions defined in [Indirect Commands Layout](#), `vkCmdProcessCommandsNVX` behaves as:

```
// For targetCommandBuffers the existing reservedSpace is reset & overwritten.

VkCommandBuffer cmd = targetCommandBuffer ?
    targetCommandBuffer.reservedSpace :
    commandBuffer;

uint32_t sequencesCount = sequencesCountBuffer ?
    min(maxSequencesCount, sequencesCountBuffer.load_uint32(sequencesCountOffset)) :
    maxSequencesCount;

cmdProcessAllSequences(cmd, objectTable,
    indirectCommandsLayout, pIndirectCommandsTokens,
    sequencesCount,
    sequencesIndexBuffer, sequencesIndexOffset);

// The stateful commands within indirectCommandsLayout will not
// affect the state of subsequent commands in the target
// command buffer (cmd)
```

*Note*



It is important to note that the state that may be affected through generated commands **must** be considered undefined for the commands following them. It is not possible to setup generated state and provoking work that uses this state outside of the generated sequence.

# Chapter 31. Sparse Resources

As documented in [Resource Memory Association](#), `VkBuffer` and `VkImage` resources in Vulkan **must** be bound completely and contiguously to a single `VkDeviceMemory` object. This binding **must** be done before the resource is used, and the binding is immutable for the lifetime of the resource.

*Sparse resources* relax these restrictions and provide these additional features:

- Sparse resources **can** be bound non-contiguously to one or more `VkDeviceMemory` allocations.
- Sparse resources **can** be re-bound to different memory allocations over the lifetime of the resource.
- Sparse resources **can** have descriptors generated and used orthogonally with memory binding commands.

## 31.1. Sparse Resource Features

Sparse resources have several features that **must** be enabled explicitly at resource creation time. The features are enabled by including bits in the `flags` parameter of `VkImageCreateInfo` or `VkBufferCreateInfo`. Each feature also has one or more corresponding feature enables specified in `VkPhysicalDeviceFeatures`.

- [Sparse binding](#) is the base feature, and provides the following capabilities:
  - Resources **can** be bound at some defined (sparse block) granularity.
  - The entire resource **must** be bound to memory before use regardless of regions actually accessed.
  - No specific mapping of image region to memory offset is defined, i.e. the location that each texel corresponds to in memory is implementation-dependent.
  - Sparse buffers have a well-defined mapping of buffer range to memory range, where an offset into a range of the buffer that is bound to a single contiguous range of memory corresponds to an identical offset within that range of memory.
  - Requested via the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` and `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` bits.
  - A sparse image created using `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` (but not `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`) supports all formats that non-sparse usage supports, and supports both `VK_IMAGE_TILING_OPTIMAL` and `VK_IMAGE_TILING_LINEAR` tiling.
- [Sparse Residency](#) builds on (and requires) the `sparseBinding` feature. It includes the following capabilities:
  - Resources do not have to be completely bound to memory before use on the device.
  - Images have a prescribed sparse image block layout, allowing specific rectangular regions of the image to be bound to specific offsets in memory allocations.
  - Consistency of access to unbound regions of the resource is defined by the absence or presence of `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict`. If this property is present, accesses to unbound regions of the resource are well defined and behave as if the

data bound is populated with all zeros; writes are discarded. When this property is absent, accesses are considered safe, but reads will return undefined values.

- Requested via the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` and `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` bits.
- Sparse residency support is advertised on a finer grain via the following features:
  - `sparseResidencyBuffer`: Support for creating `VkBuffer` objects with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidencyImage2D`: Support for creating 2D single-sampled `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidencyImage3D`: Support for creating 3D `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency2Samples`: Support for creating 2D `VkImage` objects with 2 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency4Samples`: Support for creating 2D `VkImage` objects with 4 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency8Samples`: Support for creating 2D `VkImage` objects with 8 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency16Samples`: Support for creating 2D `VkImage` objects with 16 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

Implementations supporting `sparseResidencyImage2D` are only **required** to support sparse 2D, single-sampled images. Support for sparse 3D and MSAA images is **optional** and **can** be enabled via `sparseResidencyImage3D`, `sparseResidency2Samples`, `sparseResidency4Samples`, `sparseResidency8Samples`, and `sparseResidency16Samples`.

- A sparse image created using `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` supports all non-compressed color formats with power-of-two element size that non-sparse usage supports. Additional formats **may** also be supported and **can** be queried via `vkGetPhysicalDeviceSparseImageFormatProperties`. `VK_IMAGE_TILING_LINEAR` tiling is not supported.
- [Sparse aliasing](#) provides the following capability that **can** be enabled per resource:

Allows physical memory ranges to be shared between multiple locations in the same sparse resource or between multiple sparse resources, with each binding of a memory location observing a consistent interpretation of the memory contents.

See [Sparse Memory Aliasing](#) for more information.

## 31.2. Sparse Buffers and Fully-Resident Images

Both `VkBuffer` and `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` or `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` bits **can** be thought of as a linear region of address space. In the `VkImage` case if `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` is not used, this linear region is entirely opaque, meaning that there is no application-visible mapping between texel location and memory

offset.

Unless `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` or `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` are also used, the entire resource **must** be bound to one or more `VkDeviceMemory` objects before use.

### 31.2.1. Sparse Buffer and Fully-Resident Image Block Size

The sparse block size in bytes for sparse buffers and fully-resident images is reported as `VkMemoryRequirements::alignment`. `alignment` represents both the memory alignment requirement and the binding granularity (in bytes) for sparse resources.

## 31.3. Sparse Partially-Resident Buffers

`VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` bit allow the buffer to be made only partially resident. Partially resident `VkBuffer` objects are allocated and bound identically to `VkBuffer` objects using only the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` feature. The only difference is the ability for some regions of the buffer to be unbound during device use.

## 31.4. Sparse Partially-Resident Images

`VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` bit allow specific rectangular regions of the image called sparse image blocks to be bound to specific ranges of memory. This allows the application to manage residency at either image subresource or sparse image block granularity. Each image subresource (outside of the [mip tail](#)) starts on a sparse block boundary and has dimensions that are integer multiples of the corresponding dimensions of the sparse image block.

*Note*

Applications **can** use these types of images to control LOD based on total memory consumption. If memory pressure becomes an issue the application **can** unbind and disable specific mipmap levels of images without having to recreate resources or modify texel data of unaffected levels.



The application **can** also use this functionality to access subregions of the image in a “megatexture” fashion. The application **can** create a large image and only populate the region of the image that is currently being used in the scene.

### 31.4.1. Accessing Unbound Regions

The following member of `VkPhysicalDeviceSparseProperties` affects how data in unbound regions of sparse resources are handled by the implementation:

- `residencyNonResidentStrict`

If this property is not present, reads of unbound regions of the image will return undefined values. Both reads and writes are still considered *safe* and will not affect other resources or populated regions of the image.

If this property is present, all reads of unbound regions of the image will behave as if the region was bound to memory populated with all zeros; writes will be discarded.

Formatted accesses to unbound memory **may** still alter some component values in the natural way for those accesses, e.g. substituting a value of one for alpha in formats that do not have an alpha component.

Example: Reading the alpha component of an unbacked `VK_FORMAT_R8_UNORM` image will return a value of 1.0f.

See [Physical Device Enumeration](#) for instructions for retrieving physical device properties.

### Implementor's Note

For implementations that **cannot** natively handle access to unbound regions of a resource, the implementation **may** allocate and bind memory to the unbound regions. Reads and writes to unbound regions will access the implementation-managed memory instead.

Given that the values resulting from reads of unbound regions are undefined in this scenario, implementations **may** use the same physical memory for all unbound regions of multiple resources within the same process.

#### 31.4.2. Mip Tail Regions

Sparse images created using `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` (without also using `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`) have no specific mapping of image region or image subresource to memory offset defined, so the entire image **can** be thought of as a linear opaque address region. However, images created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` do have a prescribed sparse image block layout, and hence each image subresource **must** start on a sparse block boundary. Within each array layer, the set of mip levels that have a smaller size than the sparse block size in bytes are grouped together into a *mip tail region*.

If the `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` flag is present in the `flags` member of `VkSparseImageFormatProperties`, for the image's `format`, then any mip level which has dimensions that are not integer multiples of the corresponding dimensions of the sparse image block, and all subsequent mip levels, are also included in the mip tail region.

The following member of `VkPhysicalDeviceSparseProperties` **may** affect how the implementation places mip levels in the mip tail region:

- `residencyAlignedMipSize`

Each mip tail region is bound to memory as an opaque region (i.e. **must** be bound using a `VkSparseImageOpaqueMemoryBindInfo` structure) and **may** be of a size greater than or equal to the sparse block size in bytes. This size is guaranteed to be an integer multiple of the sparse block size in bytes.

An implementation **may** choose to allow each array-layer's mip tail region to be bound to memory

independently or require that all array-layer's mip tail regions be treated as one. This is dictated by `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` in `VkSparseImageMemoryRequirements::flags`.

The following diagrams depict how `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` and `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` alter memory usage and requirements.

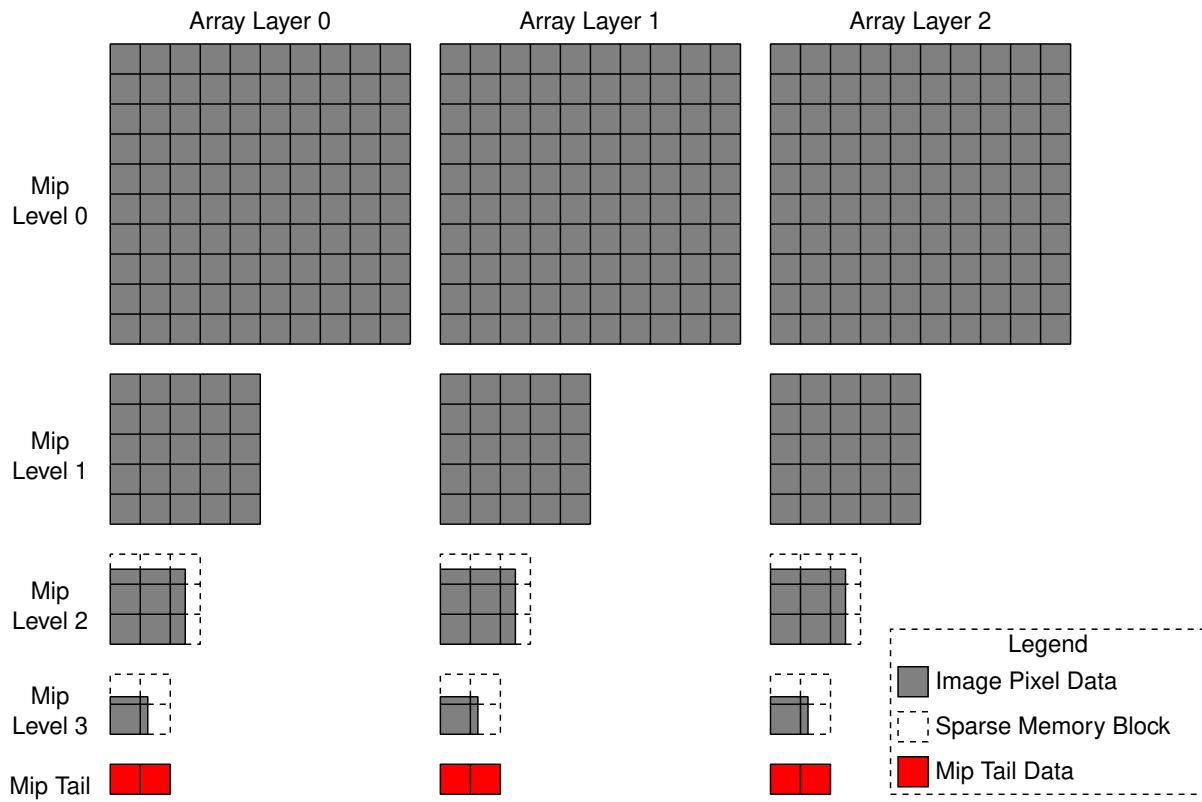


Figure 20. Sparse Image

In the absence of `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` and `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`, each array layer contains a mip tail region containing texel data for all mip levels smaller than the sparse image block in any dimension.

Mip levels that are as large or larger than a sparse image block in all dimensions **can** be bound individually. Right-edges and bottom-edges of each level are allowed to have partially used sparse blocks. Any bound partially-used-sparse-blocks **must** still have their full sparse block size in bytes allocated in memory.

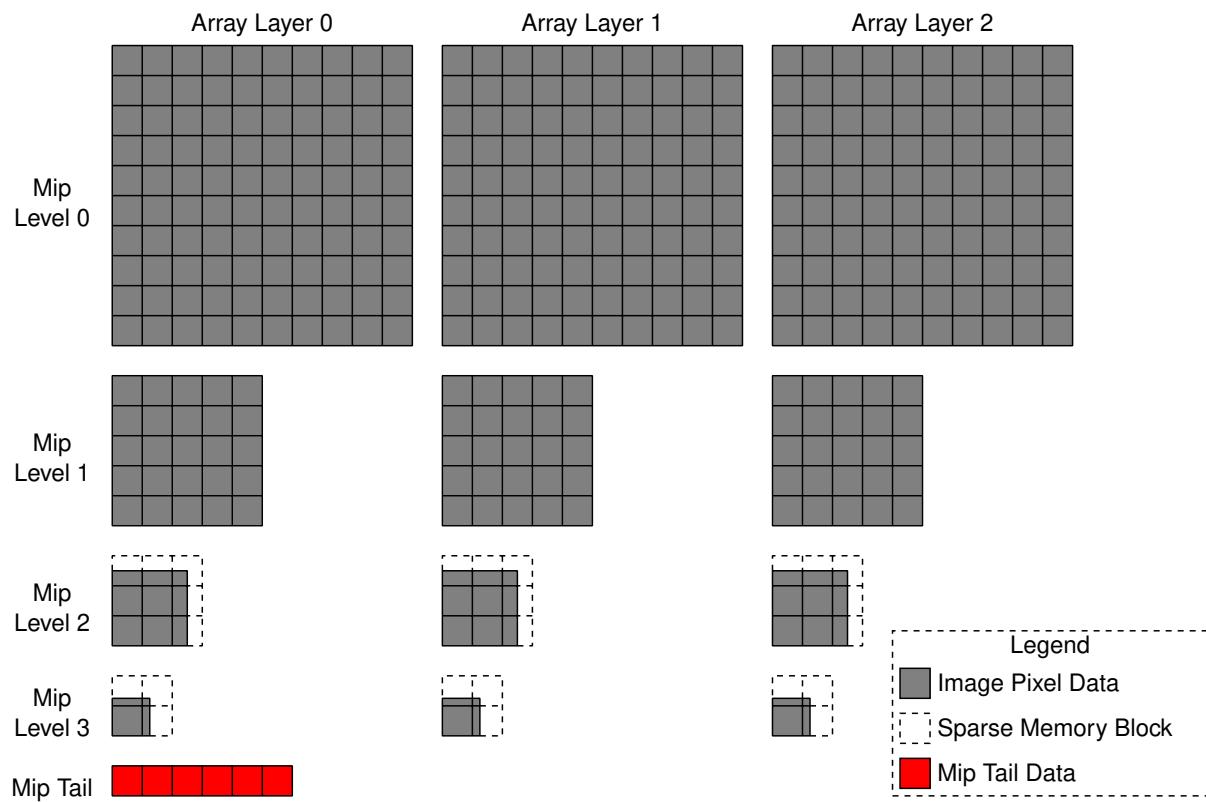


Figure 21. Sparse Image with Single Mip Tail

When `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` is present all array layers will share a single mip tail region.

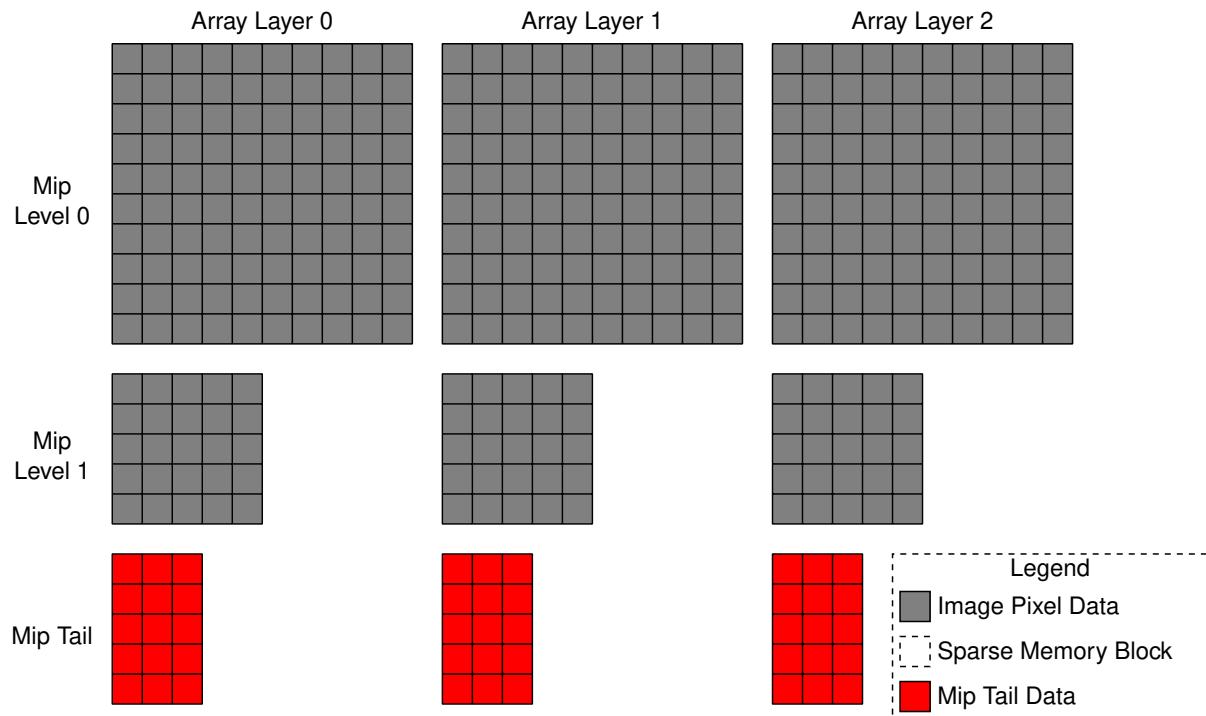


Figure 22. Sparse Image with Aligned Mip Size

#### Note



The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

When `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` is present the first mip level that would contain partially used sparse blocks begins the mip tail region. This level and all subsequent levels are placed in the mip tail. Only the first N mip levels whose dimensions are an exact multiple of the sparse image block dimensions **can** be bound and unbound on a sparse block basis.

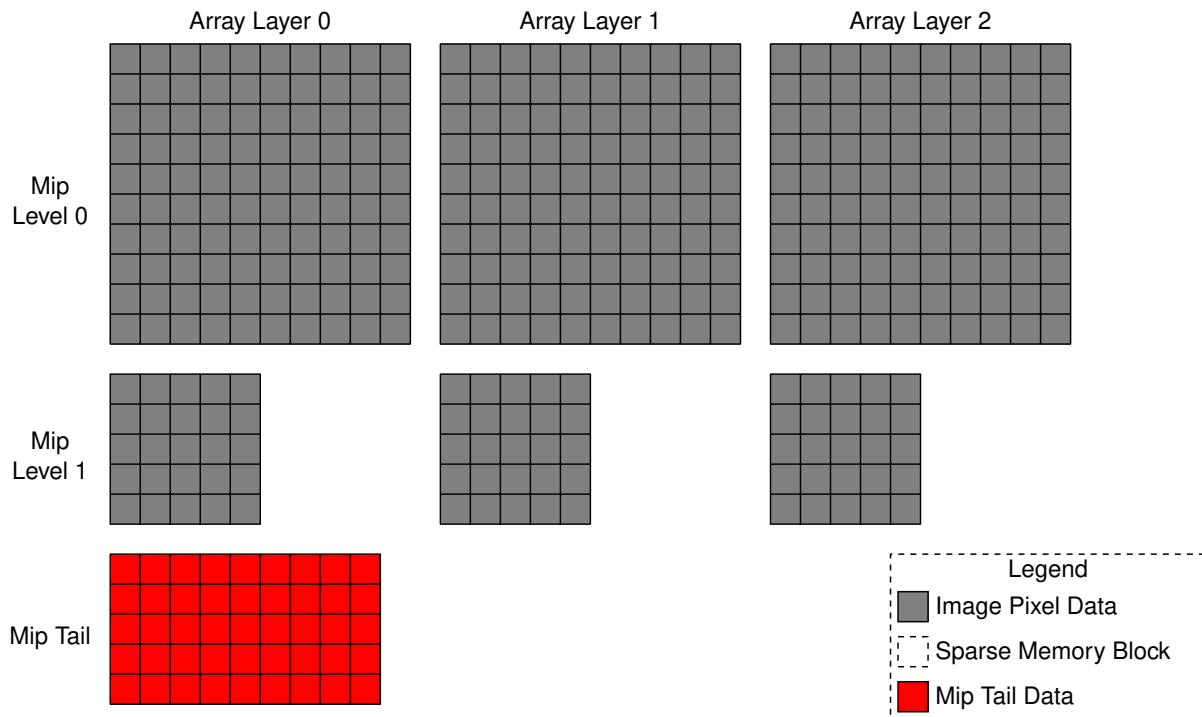


Figure 23. Sparse Image with Aligned Mip Size and Single Mip Tail

#### Note



The mip tail region is presented here in a 2D array simply for figure size reasons. It is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

When both `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` and `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` are present the constraints from each of these flags are in effect.

### 31.4.3. Standard Sparse Image Block Shapes

Standard sparse image block shapes define a standard set of dimensions for sparse image blocks that depend on the format of the image. Layout of texels or compressed texel blocks within a sparse image block is implementation dependent. All currently defined standard sparse image block shapes are 64 KB in size.

For block-compressed formats (e.g. `VK_FORMAT_BC5_UNORM_BLOCK`), the texel size is the size of the compressed texel block (e.g. 128-bit for BC5) thus the dimensions of the standard sparse image block shapes apply in terms of compressed texel blocks.

*Note*



For block-compressed formats, the dimensions of a sparse image block in terms of texels **can** be calculated by multiplying the sparse image block dimensions by the compressed texel block dimensions.

Table 41. Standard Sparse Image Block Shapes (Single Sample)

TEXEL SIZE (bits)	Block Shape (2D)	Block Shape (3D)
8-Bit	$256 \times 256 \times 1$	$64 \times 32 \times 32$
16-Bit	$256 \times 128 \times 1$	$32 \times 32 \times 32$
32-Bit	$128 \times 128 \times 1$	$32 \times 32 \times 16$
64-Bit	$128 \times 64 \times 1$	$32 \times 16 \times 16$
128-Bit	$64 \times 64 \times 1$	$16 \times 16 \times 16$

Table 42. Standard Sparse Image Block Shapes (MSAA)

TEXEL SIZE (bits)	Block Shape (2X)	Block Shape (4X)	Block Shape (8X)	Block Shape (16X)
8-Bit	$128 \times 256 \times 1$	$128 \times 128 \times 1$	$64 \times 128 \times 1$	$64 \times 64 \times 1$
16-Bit	$128 \times 128 \times 1$	$128 \times 64 \times 1$	$64 \times 64 \times 1$	$64 \times 32 \times 1$
32-Bit	$64 \times 128 \times 1$	$64 \times 64 \times 1$	$32 \times 64 \times 1$	$32 \times 32 \times 1$
64-Bit	$64 \times 64 \times 1$	$64 \times 32 \times 1$	$32 \times 32 \times 1$	$32 \times 16 \times 1$
128-Bit	$32 \times 64 \times 1$	$32 \times 32 \times 1$	$16 \times 32 \times 1$	$16 \times 16 \times 1$

Implementations that support the standard sparse image block shape for all formats listed in the [Standard Sparse Image Block Shapes \(Single Sample\)](#) and [Standard Sparse Image Block Shapes \(MSAA\)](#) tables **may** advertise the following `VkPhysicalDeviceSparseProperties`:

- `residencyStandard2DBlockShape`
- `residencyStandard2DMultisampleBlockShape`
- `residencyStandard3DBlockShape`

Reporting each of these features does *not* imply that all possible image types are supported as sparse. Instead, this indicates that no supported sparse image of the corresponding type will use custom sparse image block dimensions for any formats that have a corresponding standard sparse image block shape.

### 31.4.4. Custom Sparse Image Block Shapes

An implementation that does not support a standard image block shape for a particular sparse partially-resident image **may** choose to support a custom sparse image block shape for it instead. The dimensions of such a custom sparse image block shape are reported in `VkSparseImageFormatProperties::imageGranularity`. As with standard sparse image block shapes, the size in bytes of the custom sparse image block shape will be reported in `VkMemoryRequirements::alignment`.

Custom sparse image block dimensions are reported through `vkGetPhysicalDeviceSparseImageFormatProperties` and `vkGetImageSparseMemoryRequirements`.

An implementation **must** not support both the standard sparse image block shape and a custom sparse image block shape for the same image. The standard sparse image block shape **must** be used if it is supported.

### 31.4.5. Multiple Aspects

Partially resident images are allowed to report separate sparse properties for different aspects of the image. One example is for depth/stencil images where the implementation separates the depth and stencil data into separate planes. Another reason for multiple aspects is to allow the application to manage memory allocation for implementation-private *metadata* associated with the image. See the figure below:

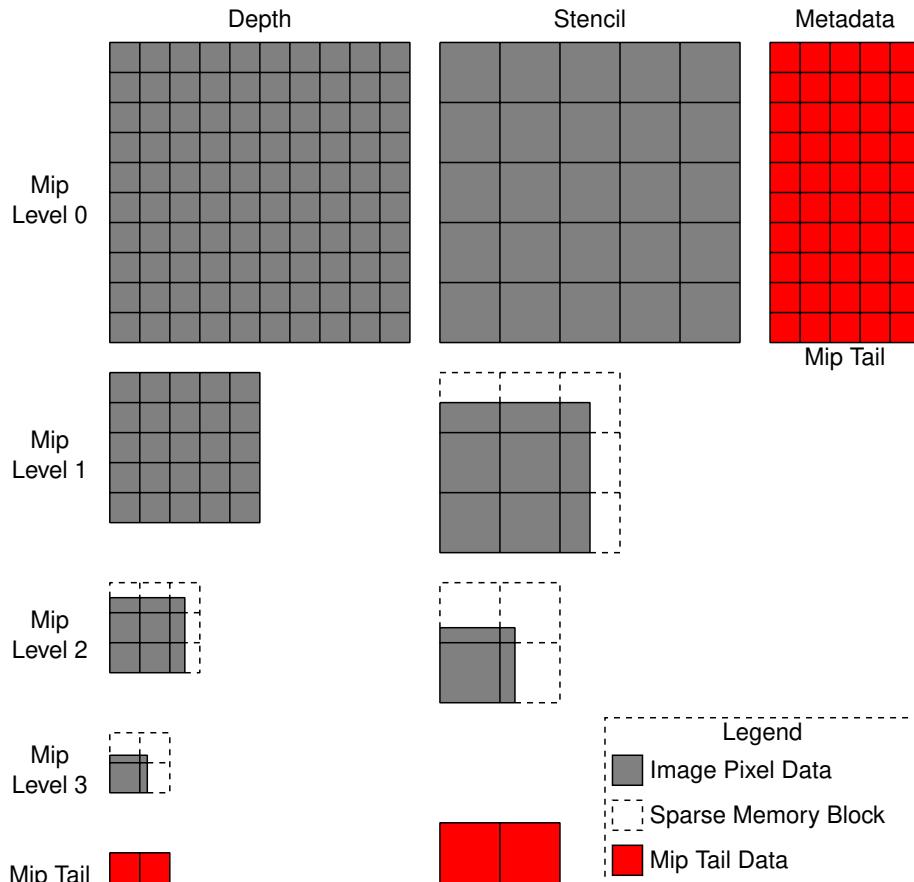


Figure 24. Multiple Aspect Sparse Image

#### Note



The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

In the figure above the depth, stencil, and metadata aspects all have unique sparse properties. The per-texel stencil data is  $\frac{1}{4}$  the size of the depth data, hence the stencil sparse blocks include  $4 \times$  the number of texels. The sparse block size in bytes for all of the aspects is identical and defined by [VkMemoryRequirements::alignment](#).

#### Metadata

The metadata aspect of an image has the following constraints:

- All metadata is reported in the mip tail region of the metadata aspect.
- All metadata **must** be bound prior to device use of the sparse image.

## 31.5. Sparse Memory Aliasing

By default sparse resources have the same aliasing rules as non-sparse resources. See [Memory Aliasing](#) for more information.

`VkDevice` objects that have the `sparseResidencyAliased` feature enabled are able to use the `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` and `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` flags for resource creation. These flags allow resources to access physical memory bound into multiple locations within one or more sparse resources in a *data consistent* fashion. This means that reading physical memory from multiple aliased locations will return the same value.

Care **must** be taken when performing a write operation to aliased physical memory. Memory dependencies **must** be used to separate writes to one alias from reads or writes to another alias. Writes to aliased memory that are not properly guarded against accesses to different aliases will have undefined results for all accesses to the aliased memory.

Applications that wish to make use of data consistent sparse memory aliasing **must** abide by the following guidelines:

- All sparse resources that are bound to aliased physical memory **must** be created with the `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` / `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` flag.
- All resources that access aliased physical memory **must** interpret the memory in the same way. This implies the following:
  - Buffers and images **cannot** alias the same physical memory in a data consistent fashion. The physical memory ranges **must** be used exclusively by buffers or used exclusively by images for data consistency to be guaranteed.
  - Memory in sparse image mip tail regions **cannot** access aliased memory in a data consistent fashion.
  - Sparse images that alias the same physical memory **must** have compatible formats and be using the same sparse image block shape in order to access aliased memory in a data consistent fashion.

Failure to follow any of the above guidelines will require the application to abide by the normal, non-sparse resource [aliasing rules](#). In this case memory **cannot** be accessed in a data consistent fashion.

*Note*



Enabling sparse resource memory aliasing **can** be a way to lower physical memory use, but it **may** reduce performance on some implementations. An application developer **can** test on their target HW and balance the memory / performance trade-offs measured.

## 31.6. Sparse Resource Implementation Guidelines

This section is Informative. It is included to aid in implementors' understanding of sparse resources.

### *Device Virtual Address*

The basic `sparseBinding` feature allows the resource to reserve its own device virtual address range at resource creation time rather than relying on a bind operation to set this. Without any other creation flags, no other constraints are relaxed compared to normal resources. All pages **must** be bound to physical memory before the device accesses the resource.

The `sparse residency` features allow sparse resources to be used even when not all pages are bound to memory. Implementations that support access to unbound pages without causing a fault **may** support `residencyNonResidentStrict`.

Not faulting on access to unbound pages is not enough to support `residencyNonResidentStrict`. An implementation **must** also guarantee that reads after writes to unbound regions of the resource always return data for the read as if the memory contains zeros. Depending on any caching hierarchy of the implementation this **may** not always be possible.

Any implementation that does not fault, but does not guarantee correct read values **must** not support `residencyNonResidentStrict`.

Any implementation that **cannot** access unbound pages without causing a fault will require the implementation to bind the entire device virtual address range to physical memory. Any pages that the application does not bind to memory **may** be bound to one (or more) “dummy” physical page(s) allocated by the implementation. Given the following properties:

- A process **must** not access memory from another process
- Reads return undefined values

It is sufficient for each host process to allocate these dummy pages and use them for all resources in that process. Implementations **may** allocate more often (per instance, per device, or per resource).

### *Binding Memory*

The byte size reported in `VkMemoryRequirements::size` **must** be greater than or equal to the amount of physical memory **required** to fully populate the resource. Some implementations require “holes” in the device virtual address range that are never accessed. These holes **may** be included in the `size` reported for the resource.

Including or not including the device virtual address holes in the resource size will alter how the implementation provides support for `VkSparseImageOpaqueMemoryBindInfo`. This operation **must** be supported for all sparse images, even ones created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

- If the holes are included in the size, this bind function becomes very easy. In most cases the `resourceOffset` is simply a device virtual address offset and the implementation can easily determine what device virtual address to bind. The cost is that the application **may** allocate more physical memory for the resource than it needs.

- If the holes are not included in the size, the application **can** allocate less physical memory than otherwise for the resource. However, in this case the implementation **must** account for the holes when mapping `resourceOffset` to the actual device virtual address intended to be mapped.

*Note*



If the application always uses `VkSparseImageMemoryBindInfo` to bind memory for the non-tail mip levels, any holes that are present in the resource size **may** never be bound.

Since `VkSparseImageMemoryBindInfo` uses texel locations to determine which device virtual addresses to bind, it is impossible to bind device virtual address holes with this operation.

#### *Binding Metadata Memory*

All metadata for sparse images have their own sparse properties and are embedded in the mip tail region for said properties. See the [Multiaspect](#) section for details.

Given that metadata is in a mip tail region, and the mip tail region **must** be reported as contiguous (either globally or per-array-layer), some implementations will have to resort to complicated offset → device virtual address mapping for handling `VkSparseImageOpaqueMemoryBindInfo`.

To make this easier on the implementation, the `VK_SPARSE_MEMORY_BIND_METADATA_BIT` explicitly specifies when metadata is bound with `VkSparseImageOpaqueMemoryBindInfo`. When this flag is not present, the `resourceOffset` **may** be treated as a strict device virtual address offset.

When `VK_SPARSE_MEMORY_BIND_METADATA_BIT` is present, the `resourceOffset` **must** have been derived explicitly from the `imageMipTailOffset` in the sparse resource properties returned for the metadata aspect. By manipulating the value returned for `imageMipTailOffset`, the `resourceOffset` does not have to correlate directly to a device virtual address offset, and **may** instead be whatever values makes it easiest for the implementation to derive the correct device virtual address.

## 31.7. Sparse Resource API

The APIs related to sparse resources are grouped into the following categories:

- [Physical Device Features](#)
- [Physical Device Sparse Properties](#)
- [Sparse Image Format Properties](#)
- [Sparse Resource Creation](#)
- [Sparse Resource Memory Requirements](#)
- [Binding Resource Memory](#)

### 31.7.1. Physical Device Features

Some sparse-resource related features are reported and enabled in `VkPhysicalDeviceFeatures`. These features **must** be supported and enabled on the `VkDevice` object before applications **can** use them. See [Physical Device Features](#) for information on how to get and set enabled device features, and for more detailed explanations of these features.

#### Sparse Physical Device Features

- `sparseBinding`: Support for creating `VkBuffer` and `VkImage` objects with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` and `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` flags, respectively.
- `sparseResidencyBuffer`: Support for creating `VkBuffer` objects with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` flag.
- `sparseResidencyImage2D`: Support for creating 2D single-sampled `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- `sparseResidencyImage3D`: Support for creating 3D `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- `sparseResidency2Samples`: Support for creating 2D `VkImage` objects with 2 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- `sparseResidency4Samples`: Support for creating 2D `VkImage` objects with 4 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- `sparseResidency8Samples`: Support for creating 2D `VkImage` objects with 8 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- `sparseResidency16Samples`: Support for creating 2D `VkImage` objects with 16 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
- `sparseResidencyAliased`: Support for creating `VkBuffer` and `VkImage` objects with the `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` and `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` flags, respectively.

### 31.7.2. Physical Device Sparse Properties

Some features of the implementation are not possible to disable, and are reported to allow applications to alter their sparse resource usage accordingly. These read-only capabilities are reported in the `VkPhysicalDeviceProperties::sparseProperties` member, which is a structure of type `VkPhysicalDeviceSparseProperties`.

The `VkPhysicalDeviceSparseProperties` structure is defined as:

```
typedef struct VkPhysicalDeviceSparseProperties {
    VkBool32    residencyStandard2DBlockShape;
    VkBool32    residencyStandard2DMultisampleBlockShape;
    VkBool32    residencyStandard3DBlockShape;
    VkBool32    residencyAlignedMipSize;
    VkBool32    residencyNonResidentStrict;
} VkPhysicalDeviceSparseProperties;
```

- `residencyStandard2DBlockShape` is `VK_TRUE` if the physical device will access all single-sample 2D sparse resources using the standard sparse image block shapes (based on image format), as described in the [Standard Sparse Image Block Shapes \(Single Sample\)](#) table. If this property is not supported the value returned in the `imageGranularity` member of the `VkSparseImageFormatProperties` structure for single-sample 2D images is not **required** to match the standard sparse image block dimensions listed in the table.
- `residencyStandard2DMultisampleBlockShape` is `VK_TRUE` if the physical device will access all multisample 2D sparse resources using the standard sparse image block shapes (based on image format), as described in the [Standard Sparse Image Block Shapes \(MSAA\)](#) table. If this property is not supported, the value returned in the `imageGranularity` member of the `VkSparseImageFormatProperties` structure for multisample 2D images is not **required** to match the standard sparse image block dimensions listed in the table.
- `residencyStandard3DBlockShape` is `VK_TRUE` if the physical device will access all 3D sparse resources using the standard sparse image block shapes (based on image format), as described in the [Standard Sparse Image Block Shapes \(Single Sample\)](#) table. If this property is not supported, the value returned in the `imageGranularity` member of the `VkSparseImageFormatProperties` structure for 3D images is not **required** to match the standard sparse image block dimensions listed in the table.
- `residencyAlignedMipSize` is `VK_TRUE` if images with mip level dimensions that are not integer multiples of the corresponding dimensions of the sparse image block **may** be placed in the mip tail. If this property is not reported, only mip levels with dimensions smaller than the `imageGranularity` member of the `VkSparseImageFormatProperties` structure will be placed in the mip tail. If this property is reported the implementation is allowed to return `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` in the `flags` member of `VkSparseImageFormatProperties`, indicating that mip level dimensions that are not integer multiples of the corresponding dimensions of the sparse image block will be placed in the mip tail.
- `residencyNonResidentStrict` specifies whether the physical device **can** consistently access non-resident regions of a resource. If this property is `VK_TRUE`, access to non-resident regions of resources will be guaranteed to return values as if the resource were populated with 0; writes to non-resident regions will be discarded.

### 31.7.3. Sparse Image Format Properties

Given that certain aspects of sparse image support, including the sparse image block dimensions, **may** be implementation-dependent, `vkGetPhysicalDeviceSparseImageFormatProperties` **can** be used to query for sparse image format properties prior to resource creation. This command is used to check whether a given set of sparse image parameters is supported and what the sparse image block shape will be.

#### Sparse Image Format Properties API

The `VkSparseImageFormatProperties` structure is defined as:

```

typedef struct VkSparseImageFormatProperties {
    VkImageAspectFlags      aspectMask;
    VkExtent3D              imageGranularity;
    VkSparseImageFormatFlags flags;
} VkSparseImageFormatProperties;

```

- **aspectMask** is a bitmask [VkImageAspectFlagBits](#) specifying which aspects of the image the properties apply to.
- **imageGranularity** is the width, height, and depth of the sparse image block in texels or compressed texel blocks.
- **flags** is a bitmask of [VkSparseImageFormatFlagBits](#) specifying additional information about the sparse resource.

Bits which **may** be set in [VkSparseImageFormatProperties::flags](#), specifying additional information about the sparse resource, are:

```

typedef enum VkSparseImageFormatFlagBits {
    VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT = 0x00000001,
    VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT = 0x00000002,
    VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT = 0x00000004,
    VK_SPARSE_IMAGE_FORMAT_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkSparseImageFormatFlagBits;

```

- **VK\_SPARSE\_IMAGE\_FORMAT\_SINGLE\_MIPTAIL\_BIT** specifies that the image uses a single mip tail region for all array layers.
- **VK\_SPARSE\_IMAGE\_FORMAT\_ALIGNED\_MIP\_SIZE\_BIT** specifies that the first mip level whose dimensions are not integer multiples of the corresponding dimensions of the sparse image block begins the mip tail region.
- **VK\_SPARSE\_IMAGE\_FORMAT\_NONSTANDARD\_BLOCK\_SIZE\_BIT** specifies that the image uses non-standard sparse image block dimensions, and the **imageGranularity** values do not match the standard sparse image block dimensions for the given format.

```
typedef VkFlags VkSparseImageFormatFlags;
```

[VkSparseImageFormatFlags](#) is a bitmask type for setting a mask of zero or more [VkSparseImageFormatFlagBits](#).

[vkGetPhysicalDeviceSparseImageFormatProperties](#) returns an array of [VkSparseImageFormatProperties](#). Each element will describe properties for one set of image aspects that are bound simultaneously in the image. This is usually one element for each aspect in the image, but for interleaved depth/stencil images there is only one element describing the combined aspects.

```

void vkGetPhysicalDeviceSparseImageFormatProperties(
    VkPhysicalDevice                                physicalDevice,
    VkFormat                                     format,
    VkImageType                                    type,
    VkSampleCountFlagBits                      samples,
    VkImageUsageFlags                           usage,
    VkImageTiling                                tiling,
    uint32_t*                                     pPropertyCount,
    VkSparseImageFormatProperties*                pProperties);

```

- **physicalDevice** is the physical device from which to query the sparse image capabilities.
- **format** is the image format.
- **type** is the dimensionality of image.
- **samples** is the number of samples per texel as defined in [VkSampleCountFlagBits](#).
- **usage** is a bitmask describing the intended usage of the image.
- **tiling** is the tiling arrangement of the texel blocks in memory.
- **pPropertyCount** is a pointer to an integer related to the number of sparse format properties available or queried, as described below.
- **pProperties** is either **NULL** or a pointer to an array of [VkSparseImageFormatProperties](#) structures.

If **pProperties** is **NULL**, then the number of sparse format properties available is returned in **pPropertyCount**. Otherwise, **pPropertyCount** **must** point to a variable set by the user to the number of elements in the **pProperties** array, and on return the variable is overwritten with the number of structures actually written to **pProperties**. If **pPropertyCount** is less than the number of sparse format properties available, at most **pPropertyCount** structures will be written.

If **VK\_IMAGE\_CREATE\_SPARSE\_RESIDENCY\_BIT** is not supported for the given arguments, **pPropertyCount** will be set to zero upon return, and no data will be written to **pProperties**.

Multiple aspects are returned for depth/stencil images that are implemented as separate planes by the implementation. The depth and stencil data planes each have unique [VkSparseImageFormatProperties](#) data.

Depth/stencil images with depth and stencil data interleaved into a single plane will return a single [VkSparseImageFormatProperties](#) structure with the **aspectMask** set to **VK\_IMAGE\_ASPECT\_DEPTH\_BIT** | **VK\_IMAGE\_ASPECT\_STENCIL\_BIT**.

## Valid Usage

- **samples** **must** be a bit value that is set in [VkImageFormatProperties::sampleCounts](#) returned by [vkGetPhysicalDeviceImageFormatProperties](#) with **format**, **type**, **tiling**, and **usage** equal to those in this command and **flags** equal to the value that is set in [VkImageCreateInfo::flags](#) when the image is created

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `format` **must** be a valid `VkFormat` value
- `type` **must** be a valid `VkImageType` value
- `samples` **must** be a valid `VkSampleCountFlagBits` value
- `usage` **must** be a valid combination of `VkImageUsageFlagBits` values
- `usage` **must** not be `0`
- `tiling` **must** be a valid `VkImageTiling` value
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkSparseImageFormatProperties` structures

`vkGetPhysicalDeviceSparseImageFormatProperties2` returns an array of `VkSparseImageFormatProperties2`. Each element will describe properties for one set of image aspects that are bound simultaneously in the image. This is usually one element for each aspect in the image, but for interleaved depth/stencil images there is only one element describing the combined aspects.

```
void vkGetPhysicalDeviceSparseImageFormatProperties2(
    VkPhysicalDevice                      physicalDevice,
    const VkPhysicalDeviceSparseImageFormatInfo2* pFormatInfo,
    uint32_t*                               pPropertyCount,
    VkSparseImageFormatProperties2*         pProperties);
```

or the equivalent command

```
void vkGetPhysicalDeviceSparseImageFormatProperties2KHR(
    VkPhysicalDevice                      physicalDevice,
    const VkPhysicalDeviceSparseImageFormatInfo2* pFormatInfo,
    uint32_t*                               pPropertyCount,
    VkSparseImageFormatProperties2*         pProperties);
```

- `physicalDevice` is the physical device from which to query the sparse image capabilities.
- `pFormatInfo` is a pointer to a `VkPhysicalDeviceSparseImageFormatInfo2` structure containing input parameters to the command.
- `pPropertyCount` is a pointer to an integer related to the number of sparse format properties available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkSparseImageFormatProperties2` structures.

`vkGetPhysicalDeviceSparseImageFormatProperties2` behaves identically to `vkGetPhysicalDeviceSparseImageFormatProperties`, with the ability to return extended information by adding extension structures to the `pNext` chain of its `pProperties` parameter.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pFormatInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceSparseImageFormatInfo2` structure
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkSparseImageFormatProperties2` structures

The `VkPhysicalDeviceSparseImageFormatInfo2` structure is defined as:

```
typedef struct VkPhysicalDeviceSparseImageFormatInfo2 {  
    VkStructureType          sType;  
    const void*              pNext;  
    VkFormat                 format;  
    VkImageType               type;  
    VkSampleCountFlagBits    samples;  
    VkImageUsageFlags        usage;  
    VkImageTiling             tiling;  
} VkPhysicalDeviceSparseImageFormatInfo2;
```

or the equivalent

```
typedef VkPhysicalDeviceSparseImageFormatInfo2  
VkPhysicalDeviceSparseImageFormatInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `format` is the image format.
- `type` is the dimensionality of image.
- `samples` is the number of samples per texel as defined in `VkSampleCountFlagBits`.
- `usage` is a bitmask describing the intended usage of the image.
- `tiling` is the tiling arrangement of the texel blocks in memory.

## Valid Usage

- **samples** **must** be a bit value that is set in `VkImageFormatProperties::sampleCounts` returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, and `usage` equal to those in this command and `flags` equal to the value that is set in `VkImageCreateInfo::flags` when the image is created

## Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2`
- **pNext** **must** be `NULL`
- **format** **must** be a valid `VkFormat` value
- **type** **must** be a valid `VkImageType` value
- **samples** **must** be a valid `VkSampleCountFlagBits` value
- **usage** **must** be a valid combination of `VkImageUsageFlagBits` values
- **usage** **must** not be `0`
- **tiling** **must** be a valid `VkImageTiling` value

The `VkSparseImageFormatProperties2` structure is defined as:

```
typedef struct VkSparseImageFormatProperties2 {
    VkStructureType             sType;
    void*                      pNext;
    VkSparseImageFormatProperties properties;
} VkSparseImageFormatProperties2;
```

or the equivalent

```
typedef VkSparseImageFormatProperties2 VkSparseImageFormatProperties2KHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **properties** is a `VkSparseImageFormatProperties` structure which is populated with the same values as in `vkGetPhysicalDeviceSparseImageFormatProperties`.

## Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2`
- **pNext** **must** be `NULL`

### 31.7.4. Sparse Resource Creation

Sparse resources require that one or more sparse feature flags be specified (as part of the `VkPhysicalDeviceFeatures` structure described previously in the [Physical Device Features](#) section) at `CreateDevice` time. When the appropriate device features are enabled, the `VK_BUFFER_CREATE_SPARSE_*` and `VK_IMAGE_CREATE_SPARSE_*` flags **can** be used. See [vkCreateBuffer](#) and [vkCreateImage](#) for details of the resource creation APIs.

*Note*



Specifying `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` or `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` requires specifying `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` or `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, respectively, as well. This means that resources **must** be created with the appropriate `*_SPARSE_BINDING_BIT` to be used with the sparse binding command (`vkQueueBindSparse`).

### 31.7.5. Sparse Resource Memory Requirements

Sparse resources have specific memory requirements related to binding sparse memory. These memory requirements are reported differently for `VkBuffer` objects and `VkImage` objects.

#### Buffer and Fully-Resident Images

Buffers (both fully and partially resident) and fully-resident images **can** be bound to memory using only the data from `VkMemoryRequirements`. For all sparse resources the `VkMemoryRequirements::alignment` member specifies both the bindable sparse block size in bytes and **required** alignment of `VkDeviceMemory`.

#### Partially Resident Images

Partially resident images have a different method for binding memory. As with buffers and fully resident images, the `VkMemoryRequirements::alignment` field specifies the bindable sparse block size in bytes for the image.

Requesting sparse memory requirements for `VkImage` objects using `vkGetImageSparseMemoryRequirements` will return an array of one or more `VkSparseImageMemoryRequirements` structures. Each structure describes the sparse memory requirements for a group of aspects of the image.

The sparse image **must** have been created using the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag to retrieve valid sparse image memory requirements.

#### Sparse Image Memory Requirements

The `VkSparseImageMemoryRequirements` structure is defined as:

```

typedef struct VkSparseImageMemoryRequirements {
    VkSparseImageFormatProperties      formatProperties;
    uint32_t                           imageMipTailFirstLod;
    VkDeviceSize                      imageMipTailSize;
    VkDeviceSize                      imageMipTailOffset;
    VkDeviceSize                      imageMipTailStride;
} VkSparseImageMemoryRequirements;

```

- `formatProperties.aspectMask` is the set of aspects of the image that this sparse memory requirement applies to. This will usually have a single aspect specified. However, depth/stencil images **may** have depth and stencil data interleaved in the same sparse block, in which case both `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT` would be present.
- `formatProperties.imageGranularity` describes the dimensions of a single bindable sparse image block in texel units. For aspect `VK_IMAGE_ASPECT_METADATA_BIT`, all dimensions will be zero. All metadata is located in the mip tail region.
- `formatProperties.flags` is a bitmask of `VkSparseImageFormatFlagBits`:
  - If `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` is set the image uses a single mip tail region for all array layers.
  - If `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` is set the dimensions of mip levels **must** be integer multiples of the corresponding dimensions of the sparse image block for levels not located in the mip tail.
  - If `VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT` is set the image uses non-standard sparse image block dimensions. The `formatProperties.imageGranularity` values do not match the standard sparse image block dimension corresponding to the image's format.
- `imageMipTailFirstLod` is the first mip level at which image subresources are included in the mip tail region.
- `imageMipTailSize` is the memory size (in bytes) of the mip tail region. If `formatProperties.flags` contains `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`, this is the size of the whole mip tail, otherwise this is the size of the mip tail of a single array layer. This value is guaranteed to be a multiple of the sparse block size in bytes.
- `imageMipTailOffset` is the opaque memory offset used with `VkSparseImageOpaqueMemoryBindInfo` to bind the mip tail region(s).
- `imageMipTailStride` is the offset stride between each array-layer's mip tail, if `formatProperties.flags` does not contain `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` (otherwise the value is undefined).

To query sparse memory requirements for an image, call:

```

void vkGetImageSparseMemoryRequirements(
    VkDevice                           device,
    VkImage                            image,
    uint32_t*                          pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements*   pSparseMemoryRequirements);

```

- `device` is the logical device that owns the image.
- `image` is the `VkImage` object to get the memory requirements for.
- `pSparseMemoryRequirementCount` is a pointer to an integer related to the number of sparse memory requirements available or queried, as described below.
- `pSparseMemoryRequirements` is either `NULL` or a pointer to an array of `VkSparseImageMemoryRequirements` structures.

If `pSparseMemoryRequirements` is `NULL`, then the number of sparse memory requirements available is returned in `pSparseMemoryRequirementCount`. Otherwise, `pSparseMemoryRequirementCount` **must** point to a variable set by the user to the number of elements in the `pSparseMemoryRequirements` array, and on return the variable is overwritten with the number of structures actually written to `pSparseMemoryRequirements`. If `pSparseMemoryRequirementCount` is less than the number of sparse memory requirements available, at most `pSparseMemoryRequirementCount` structures will be written.

If the image was not created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` then `pSparseMemoryRequirementCount` will be set to zero and `pSparseMemoryRequirements` will not be written to.

#### Note



It is legal for an implementation to report a larger value in `VkMemoryRequirements::size` than would be obtained by adding together memory sizes for all `VkSparseImageMemoryRequirements` returned by `vkGetImageSparseMemoryRequirements`. This **may** occur when the implementation requires unused padding in the address range describing the resource.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `image` **must** be a valid `VkImage` handle
- `pSparseMemoryRequirementCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pSparseMemoryRequirementCount` is not `0`, and `pSparseMemoryRequirements` is not `NULL`, `pSparseMemoryRequirements` **must** be a valid pointer to an array of `pSparseMemoryRequirementCount` `VkSparseImageMemoryRequirements` structures
- `image` **must** have been created, allocated, or retrieved from `device`

To query sparse memory requirements for an image, call:

```
void vkGetImageSparseMemoryRequirements2(
    VkDevice                               device,
    const VkImageSparseMemoryRequirementsInfo2* pInfo,
    uint32_t*                                pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements2*         pSparseMemoryRequirements);
```

or the equivalent command

```
void vkGetImageSparseMemoryRequirements2KHR(  
    VkDevice device,  
    const VkImageSparseMemoryRequirementsInfo2* pInfo,  
    uint32_t* pSparseMemoryRequirementCount,  
    VkSparseImageMemoryRequirements2* pSparseMemoryRequirements);
```

- `device` is the logical device that owns the image.
- `pInfo` is a pointer to a `VkImageSparseMemoryRequirementsInfo2` structure containing parameters required for the memory requirements query.
- `pSparseMemoryRequirementCount` is a pointer to an integer related to the number of sparse memory requirements available or queried, as described below.
- `pSparseMemoryRequirements` is either `NULL` or a pointer to an array of `VkSparseImageMemoryRequirements2` structures.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pInfo` **must** be a valid pointer to a valid `VkImageSparseMemoryRequirementsInfo2` structure
- `pSparseMemoryRequirementCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pSparseMemoryRequirementCount` is not `0`, and `pSparseMemoryRequirements` is not `NULL`, `pSparseMemoryRequirements` **must** be a valid pointer to an array of `pSparseMemoryRequirementCount` `VkSparseImageMemoryRequirements2` structures

The `VkImageSparseMemoryRequirementsInfo2` structure is defined as:

```
typedef struct VkImageSparseMemoryRequirementsInfo2 {  
    VkStructureType sType;  
    const void* pNext;  
    VkImage image;  
} VkImageSparseMemoryRequirementsInfo2;
```

or the equivalent

```
typedef VkImageSparseMemoryRequirementsInfo2 VkImageSparseMemoryRequirementsInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `image` is the image to query.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2`
- `pNext` **must** be `NULL`
- `image` **must** be a valid `VkImage` handle

The `VkSparseImageMemoryRequirements2` structure is defined as:

```
typedef struct VkSparseImageMemoryRequirements2 {
    VkStructureType          sType;
    void*                    pNext;
    VkSparseImageMemoryRequirements memoryRequirements;
} VkSparseImageMemoryRequirements2;
```

or the equivalent

```
typedef VkSparseImageMemoryRequirements2 VkSparseImageMemoryRequirements2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memoryRequirements` is a `VkSparseImageMemoryRequirements` structure describing the memory requirements of the sparse image.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2`
- `pNext` **must** be `NULL`

### 31.7.6. Binding Resource Memory

Non-sparse resources are backed by a single physical allocation prior to device use (via `vkBindImageMemory` or `vkBindBufferMemory`), and their backing **must** not be changed. On the other hand, sparse resources **can** be bound to memory non-contiguously and these bindings **can** be altered during the lifetime of the resource.

#### Note



It is important to note that freeing a `VkDeviceMemory` object with `vkFreeMemory` will not cause resources (or resource regions) bound to the memory object to become unbound. Applications **must** not access resources bound to memory that has been freed.

Sparse memory bindings execute on a queue that includes the `VK_QUEUE_SPARSE_BINDING_BIT` bit.

Applications **must** use [synchronization primitives](#) to guarantee that other queues do not access ranges of memory concurrently with a binding change. Applications **can** access other ranges of the same resource while a bind operation is executing.

*Note*



Implementations **must** provide a guarantee that simultaneously binding sparse blocks while another queue accesses those same sparse blocks via a sparse resource **must** not access memory owned by another process or otherwise corrupt the system.

While some implementations **may** include `VK_QUEUE_SPARSE_BINDING_BIT` support in queue families that also include graphics and compute support, other implementations **may** only expose a `VK_QUEUE_SPARSE_BINDING_BIT`-only queue family. In either case, applications **must** use [synchronization primitives](#) to explicitly request any ordering dependencies between sparse memory binding operations and other graphics/compute/transfer operations, as sparse binding operations are not automatically ordered against command buffer execution, even within a single queue.

When binding memory explicitly for the `VK_IMAGE_ASPECT_METADATA_BIT` the application **must** use the `VK_SPARSE_MEMORY_BIND_METADATA_BIT` in the `VkSparseMemoryBind::flags` field when binding memory. Binding memory for metadata is done the same way as binding memory for the mip tail, with the addition of the `VK_SPARSE_MEMORY_BIND_METADATA_BIT` flag.

Binding the mip tail for any aspect **must** only be performed using `VkSparseImageOpaqueMemoryBindInfo`. If `formatProperties.flags` contains `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`, then it **can** be bound with a single `VkSparseMemoryBind` structure, with `resourceOffset = imageMipTailOffset` and `size = imageMipTailSize`.

If `formatProperties.flags` does not contain `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` then the offset for the mip tail in each array layer is given as:

```
arrayMipTailOffset = imageMipTailOffset + arrayLayer * imageMipTailStride;
```

and the mip tail **can** be bound with `layerCount` `VkSparseMemoryBind` structures, each using `size = imageMipTailSize` and `resourceOffset = arrayMipTailOffset` as defined above.

Sparse memory binding is handled by the following APIs and related data structures.

## Sparse Memory Binding Functions

The `VkSparseMemoryBind` structure is defined as:

```

typedef struct VkSparseMemoryBind {
    VkDeviceSize          resourceOffset;
    VkDeviceSize          size;
    VkDeviceMemory        memory;
    VkDeviceSize          memoryOffset;
    VkSparseMemoryBindFlags flags;
} VkSparseMemoryBind;

```

- `resourceOffset` is the offset into the resource.
- `size` is the size of the memory region to be bound.
- `memory` is the `VkDeviceMemory` object that the range of the resource is bound to. If `memory` is `VK_NULL_HANDLE`, the range is unbound.
- `memoryOffset` is the offset into the `VkDeviceMemory` object to bind the resource range to. If `memory` is `VK_NULL_HANDLE`, this value is ignored.
- `flags` is a bitmask of `VkSparseMemoryBindFlagBits` specifying usage of the binding operation.

The *binding range* [`resourceOffset`, `resourceOffset + size`] has different constraints based on `flags`. If `flags` contains `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range **must** be within the mip tail region of the metadata aspect. This metadata region is defined by:

`metadataRegion = [base, base + imageMipTailSize)`

`base = imageMipTailOffset + imageMipTailStride × n`

and `imageMipTailOffset`, `imageMipTailSize`, and `imageMipTailStride` values are from the `VkSparseImageMemoryRequirements` corresponding to the metadata aspect of the image, and `n` is a valid array layer index for the image,

`imageMipTailStride` is considered to be zero for aspects where `VkSparseImageMemoryRequirements::formatProperties.flags` contains `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`.

If `flags` does not contain `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range **must** be within the range [0, `VkMemoryRequirements::size`).

## Valid Usage

- If `memory` is not `VK_NULL_HANDLE`, `memory` and `memoryOffset` **must** match the memory requirements of the resource, as described in section [Resource Memory Association](#)
- If `memory` is not `VK_NULL_HANDLE`, `memory` **must** not have been created with a memory type that reports `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit set
- `size` **must** be greater than `0`
- `resourceOffset` **must** be less than the size of the resource
- `size` **must** be less than or equal to the size of the resource minus `resourceOffset`
- `memoryOffset` **must** be less than the size of `memory`
- `size` **must** be less than or equal to the size of `memory` minus `memoryOffset`
- If `memory` was created with `VkExportMemoryAllocateInfo::handleTypes` not equal to `0`, at least one handle type it contained **must** also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` or `VkExternalMemoryImageCreateInfo::handleTypes` when the resource was created.
- If `memory` was created by a memory import operation, the external handle type of the imported memory **must** also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` or `VkExternalMemoryImageCreateInfo::handleTypes` when the resource was created.

## Valid Usage (Implicit)

- If `memory` is not `VK_NULL_HANDLE`, `memory` **must** be a valid `VkDeviceMemory` handle
- `flags` **must** be a valid combination of `VkSparseMemoryBindFlagBits` values

Bits which **can** be set in `VkSparseMemoryBind::flags`, specifying usage of a sparse memory binding operation, are:

```
typedef enum VkSparseMemoryBindFlagBits {
    VK_SPARSE_MEMORY_BIND_METADATA_BIT = 0x00000001,
    VK_SPARSE_MEMORY_BIND_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkSparseMemoryBindFlagBits;
```

- `VK_SPARSE_MEMORY_BIND_METADATA_BIT` specifies that the memory being bound is only for the metadata aspect.

```
typedef VkFlags VkSparseMemoryBindFlags;
```

`VkSparseMemoryBindFlags` is a bitmask type for setting a mask of zero or more `VkSparseMemoryBindFlagBits`.

Memory is bound to `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` flag using the following structure:

```
typedef struct VkSparseBufferMemoryBindInfo {
    VkBuffer                  buffer;
    uint32_t                  bindCount;
    const VkSparseMemoryBind* pBinds;
} VkSparseBufferMemoryBindInfo;
```

- `buffer` is the `VkBuffer` object to be bound.
- `bindCount` is the number of `VkSparseMemoryBind` structures in the `pBinds` array.
- `pBinds` is a pointer to array of `VkSparseMemoryBind` structures.

### Valid Usage (Implicit)

- `buffer` **must** be a valid `VkBuffer` handle
- `pBinds` **must** be a valid pointer to an array of `bindCount` valid `VkSparseMemoryBind` structures
- `bindCount` **must** be greater than 0

Memory is bound to opaque regions of `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` flag using the following structure:

```
typedef struct VkSparseImageOpaqueMemoryBindInfo {
    VkImage                  image;
    uint32_t                  bindCount;
    const VkSparseMemoryBind* pBinds;
} VkSparseImageOpaqueMemoryBindInfo;
```

- `image` is the `VkImage` object to be bound.
- `bindCount` is the number of `VkSparseMemoryBind` structures in the `pBinds` array.
- `pBinds` is a pointer to an array of `VkSparseMemoryBind` structures.

### Valid Usage

- If the `flags` member of any element of `pBinds` contains `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range defined **must** be within the mip tail region of the metadata aspect of `image`

## Valid Usage (Implicit)

- **image** **must** be a valid `VkImage` handle
- **pBinds** **must** be a valid pointer to an array of **bindCount** valid `VkSparseMemoryBind` structures
- **bindCount** **must** be greater than **0**

### Note

This operation is normally used to bind memory to fully-resident sparse images or for mip tail regions of partially resident images. However, it **can** also be used to bind memory for the entire binding range of partially resident images.

In case **flags** does not contain `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the **resourceOffset** is in the range [0, `VkMemoryRequirements::size`), This range includes data from all aspects of the image, including metadata. For most implementations this will probably mean that the **resourceOffset** is a simple device address offset within the resource. It is possible for an application to bind a range of memory that includes both resource data and metadata. However, the application would not know what part of the image the memory is used for, or if any range is being used for metadata.

When **flags** contains `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range specified **must** be within the mip tail region of the metadata aspect. In this case the **resourceOffset** is not **required** to be a simple device address offset within the resource. However, it *is* defined to be within [`imageMipTailOffset`, `imageMipTailOffset` + `imageMipTailSize`) for the metadata aspect. See `VkSparseMemoryBind` for the full constraints on binding region with this flag present.

Memory **can** be bound to sparse image blocks of `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag using the following structure:

```
typedef struct VkSparseImageMemoryBindInfo {
    VkImage                  image;
    uint32_t                 bindCount;
    const VkSparseImageMemoryBind* pBinds;
} VkSparseImageMemoryBindInfo;
```

- **image** is the `VkImage` object to be bound
- **bindCount** is the number of `VkSparseImageMemoryBind` structures in **pBinds** array
- **pBinds** is a pointer to an array of `VkSparseImageMemoryBind` structures

## Valid Usage

- The `subresource.mipLevel` member of each element of `pBinds` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created
- The `subresource.arrayLayer` member of each element of `pBinds` **must** be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created

## Valid Usage (Implicit)

- `image` **must** be a valid `VkImage` handle
- `pBinds` **must** be a valid pointer to an array of `bindCount` valid `VkSparseImageMemoryBind` structures
- `bindCount` **must** be greater than `0`

The `VkSparseImageMemoryBind` structure is defined as:

```
typedef struct VkSparseImageMemoryBind {  
    VkImageSubresource      subresource;  
    VkOffset3D              offset;  
    VkExtent3D              extent;  
    VkDeviceMemory          memory;  
    VkDeviceSize             memoryOffset;  
    VkSparseMemoryBindFlags flags;  
} VkSparseImageMemoryBind;
```

- `subresource` is the image *aspect* and region of interest in the image.
- `offset` are the coordinates of the first texel within the image subresource to bind.
- `extent` is the size in texels of the region within the image subresource to bind. The extent **must** be a multiple of the sparse image block dimensions, except when binding sparse image blocks along the edge of an image subresource it **can** instead be such that any coordinate of `offset` + `extent` equals the corresponding dimensions of the image subresource.
- `memory` is the `VkDeviceMemory` object that the sparse image blocks of the image are bound to. If `memory` is `VK_NULL_HANDLE`, the sparse image blocks are unbound.
- `memoryOffset` is an offset into `VkDeviceMemory` object. If `memory` is `VK_NULL_HANDLE`, this value is ignored.
- `flags` are sparse memory binding flags.

## Valid Usage

- If the [sparse aliased residency](#) feature is not enabled, and if any other resources are bound to ranges of [memory](#), the range of [memory](#) being bound **must** not overlap with those bound ranges
- [memory](#) and [memoryOffset](#) **must** match the memory requirements of the calling command's [image](#), as described in section [Resource Memory Association](#)
- [subresource](#) **must** be a valid image subresource for [image](#) (see [Image Views](#))
- [offset.x](#) **must** be a multiple of the sparse image block width ([VkSparseImageFormatProperties::imageGranularity.width](#)) of the image
- [extent.width](#) **must** either be a multiple of the sparse image block width of the image, or else ([extent.width + offset.x](#)) **must** equal the width of the image subresource
- [offset.y](#) **must** be a multiple of the sparse image block height ([VkSparseImageFormatProperties::imageGranularity.height](#)) of the image
- [extent.height](#) **must** either be a multiple of the sparse image block height of the image, or else ([extent.height + offset.y](#)) **must** equal the height of the image subresource
- [offset.z](#) **must** be a multiple of the sparse image block depth ([VkSparseImageFormatProperties::imageGranularity.depth](#)) of the image
- [extent.depth](#) **must** either be a multiple of the sparse image block depth of the image, or else ([extent.depth + offset.z](#)) **must** equal the depth of the image subresource
- If [memory](#) was created with [VkExportMemoryAllocateInfo::handleTypes](#) not equal to [0](#), at least one handle type it contained **must** also have been set in [VkExternalMemoryImageCreateInfo::handleTypes](#) when the image was created.
- If [memory](#) was created by a memory import operation, the external handle type of the imported memory **must** also have been set in [VkExternalMemoryImageCreateInfo::handleTypes](#) when [image](#) was created.

## Valid Usage (Implicit)

- [subresource](#) **must** be a valid [VkImageSubresource](#) structure
- If [memory](#) is not [VK\\_NULL\\_HANDLE](#), [memory](#) **must** be a valid [VkDeviceMemory](#) handle
- [flags](#) **must** be a valid combination of [VkSparseMemoryBindFlagBits](#) values

To submit sparse binding operations to a queue, call:

```
VkResult vkQueueBindSparse(  
    VkQueue                           queue,  
    uint32_t                          bindInfoCount,  
    const VkBindSparseInfo*           pBindInfo,  
    VkFence);
```

- `queue` is the queue that the sparse binding operations will be submitted to.
- `bindInfoCount` is the number of elements in the `pBindInfo` array.
- `pBindInfo` is a pointer to an array of `VkBindSparseInfo` structures, each specifying a sparse binding submission batch.
- `fence` is an **optional** handle to a fence to be signaled. If `fence` is not `VK_NULL_HANDLE`, it defines a [fence signal operation](#).

`vkQueueBindSparse` is a [queue submission command](#), with each batch defined by an element of `pBindInfo` as an instance of the `VkBindSparseInfo` structure. Batches begin execution in the order they appear in `pBindInfo`, but **may** complete out of order.

Within a batch, a given range of a resource **must** not be bound more than once. Across batches, if a range is to be bound to one allocation and offset and then to another allocation and offset, then the application **must** guarantee (usually using semaphores) that the binding operations are executed in the correct order, as well as to order binding operations against the execution of command buffer submissions.

As no operation to `vkQueueBindSparse` causes any pipeline stage to access memory, synchronization primitives used in this command effectively only define execution dependencies.

Additional information about fence and semaphore operation is described in [the synchronization chapter](#).

## Valid Usage

- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be unsignaled
- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** not be associated with any other queue command that has not yet completed execution on that queue
- Each element of the `pSignalSemaphores` member of each element of `pBindInfo` **must** be unsignaled when the semaphore signal operation it defines is executed on the device
- When a semaphore wait operation referring to a binary semaphore defined by any element of the `pWaitSemaphores` member of any element of `pBindInfo` executes on `queue`, there **must** be no other queues waiting on the same semaphore.
- All elements of the `pWaitSemaphores` member of all elements of `pBindInfo` member referring to a binary semaphore **must** be semaphores that are signaled, or have [semaphore signal operations](#) previously submitted for execution.
- All elements of the `pWaitSemaphores` member of all elements of `pBindInfo` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR` **must** reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) **must** have also been submitted for execution.

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- If `bindInfoCount` is not `0`, `pBindInfo` **must** be a valid pointer to an array of `bindInfoCount` valid `VkBIndSparseInfo` structures
- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be a valid `VkFence` handle
- The `queue` **must** support sparse binding operations
- Both of `fence`, and `queue` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `queue` **must** be externally synchronized
- Host access to `pBindInfo[]`.`pBufferBinds[]`.`buffer` **must** be externally synchronized
- Host access to `pBindInfo[]`.`pImageOpaqueBinds[]`.`image` **must** be externally synchronized
- Host access to `pBindInfo[]`.`pImageBinds[]`.`image` **must** be externally synchronized
- Host access to `fence` **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	SPARSE_BINDING	-

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

The `VkBIndSparseInfo` structure is defined as:

```

typedef struct VkBindSparseInfo {
    VkStructureType
    const void*
    uint32_t
    const VkSemaphore*
    uint32_t
    const VkSparseBufferMemoryBindInfo*
    uint32_t
    const VkSparseImageOpaqueMemoryBindInfo*
    uint32_t
    const VkSparseImageMemoryBindInfo*
    uint32_t
    const VkSemaphore*
} VkBindSparseInfo;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **waitSemaphoreCount** is the number of semaphores upon which to wait before executing the sparse binding operations for the batch.
- **pWaitSemaphores** is a pointer to an array of semaphores upon which to wait on before the sparse binding operations for this batch begin execution. If semaphores to wait on are provided, they define a [semaphore wait operation](#).
- **bufferBindCount** is the number of sparse buffer bindings to perform in the batch.
- **pBufferBinds** is a pointer to an array of [VkSparseBufferMemoryBindInfo](#) structures.
- **imageOpaqueBindCount** is the number of opaque sparse image bindings to perform.
- **pImageOpaqueBinds** is a pointer to an array of [VkSparseImageOpaqueMemoryBindInfo](#) structures, indicating opaque sparse image bindings to perform.
- **imageBindCount** is the number of sparse image bindings to perform.
- **pImageBinds** is a pointer to an array of [VkSparseImageMemoryBindInfo](#) structures, indicating sparse image bindings to perform.
- **signalSemaphoreCount** is the number of semaphores to be signaled once the sparse binding operations specified by the structure have completed execution.
- **pSignalSemaphores** is a pointer to an array of semaphores which will be signaled when the sparse binding operations for this batch have completed execution. If semaphores to be signaled are provided, they define a [semaphore signal operation](#).

## Valid Usage

- If any element of `pWaitSemaphores` or `pSignalSemaphores` was created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` then the `pNext` chain **must** include an instance of `VkTimelineSemaphoreSubmitInfoKHR`
- If the `pNext` chain of this structure includes an instance of `VkTimelineSemaphoreSubmitInfoKHR` and any element of `pWaitSemaphores` was created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` then its `waitSemaphoreValueCount` member **must** equal `waitForSemaphoreCount`
- If the `pNext` chain of this structure includes an instance of `VkTimelineSemaphoreSubmitInfoKHR` and any element of `pSignalSemaphores` was created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` then its `signalSemaphoreValueCount` member **must** equal `signalSemaphoreCount`
- For each element of `pSignalSemaphores` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` the corresponding element of `VkTimelineSemaphoreSubmitInfoKHR::pSignalSemaphoreValues` **must** have a value greater than the current value of the semaphore when the `semaphore signal operation` is executed
- For each element of `pWaitSemaphores` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` the corresponding element of `VkTimelineSemaphoreSubmitInfoKHR::pWaitSemaphoreValues` **must** have a value which does not differ from the current value of the semaphore or from the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.
- For each element of `pSignalSemaphores` created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` the corresponding element of `VkTimelineSemaphoreSubmitInfoKHR::pSignalSemaphoreValues` **must** have a value which does not differ from the current value of the semaphore or from the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_BIND_SPARSE_INFO`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDeviceGroupBindSparseInfo` or `VkTimelineSemaphoreSubmitInfoKHR`
- Each `sType` member in the `pNext` chain must be unique
- If `waitSemaphoreCount` is not `0`, `pWaitSemaphores` must be a valid pointer to an array of `waitSemaphoreCount` valid `VkSemaphore` handles
- If `bufferBindCount` is not `0`, `pBufferBinds` must be a valid pointer to an array of `bufferBindCount` valid `VkSparseBufferMemoryBindInfo` structures
- If `imageOpaqueBindCount` is not `0`, `pImageOpaqueBinds` must be a valid pointer to an array of `imageOpaqueBindCount` valid `VkSparseImageOpaqueMemoryBindInfo` structures
- If `imageBindCount` is not `0`, `pImageBinds` must be a valid pointer to an array of `imageBindCount` valid `VkSparseImageMemoryBindInfo` structures
- If `signalSemaphoreCount` is not `0`, `pSignalSemaphores` must be a valid pointer to an array of `signalSemaphoreCount` valid `VkSemaphore` handles
- Both of the elements of `pSignalSemaphores`, and the elements of `pWaitSemaphores` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

To specify the values to use when waiting for and signaling semaphores created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR`, add the `VkTimelineSemaphoreSubmitInfoKHR` structure to the `pNext` chain of the `VkBindSparseInfo` structure.

If the `pNext` chain of `VkBindSparseInfo` includes a `VkDeviceGroupBindSparseInfo` structure, then that structure includes device indices specifying which instance of the resources and memory are bound.

The `VkDeviceGroupBindSparseInfo` structure is defined as:

```
typedef struct VkDeviceGroupBindSparseInfo {  
    VkStructureType    sType;  
    const void*       pNext;  
    uint32_t          resourceDeviceIndex;  
    uint32_t          memoryDeviceIndex;  
} VkDeviceGroupBindSparseInfo;
```

or the equivalent

```
typedef VkDeviceGroupBindSparseInfo VkDeviceGroupBindSparseInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `resourceDeviceIndex` is a device index indicating which instance of the resource is bound.
- `memoryDeviceIndex` is a device index indicating which instance of the memory the resource instance is bound to.

These device indices apply to all buffer and image memory binds included in the batch pointing to this structure. The semaphore waits and signals for the batch are executed only by the physical device specified by the `resourceDeviceIndex`.

If this structure is not present, `resourceDeviceIndex` and `memoryDeviceIndex` are assumed to be zero.

### Valid Usage

- `resourceDeviceIndex` and `memoryDeviceIndex` **must** both be valid device indices.
- Each memory allocation bound in this batch **must** have allocated an instance for `memoryDeviceIndex`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO`

## 31.8. Examples

The following examples illustrate basic creation of sparse images and binding them to physical memory.

### 31.8.1. Basic Sparse Resources

This basic example creates a normal `VkImage` object but uses fine-grained memory allocation to back the resource with multiple memory ranges.

```

VkDevice           device;
VkQueue            queue;
VkImage            sparseImage;
VkAllocationCallbacks* pAllocator = NULL;
VkMemoryRequirements   memoryRequirements = {};
VkDeviceSize        offset = 0;
VkSparseMemoryBind  binds[MAX_CHUNKS] = {}; // MAX_CHUNKS is NOT part of Vulkan
uint32_t            bindCount = 0;

// ...

// Allocate image object
const VkImageCreateInfo sparseImageInfo =

```

```

{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,           // sType
    NULL,                                         // pNext
    VK_IMAGE_CREATE_SPARSE_BINDING_BIT | ...,     // flags
    ...
};

vkCreateImage(device, &sparseImageInfo, pAllocator, &sparseImage);

// Get memory requirements
vkGetImageMemoryRequirements(
    device,
    sparseImage,
    &memoryRequirements);

// Bind memory in fine-grained fashion, find available memory ranges
// from potentially multiple VkDeviceMemory pools.
// (Illustration purposes only, can be optimized for perf)
while (memoryRequirements.size && bindCount < MAX_CHUNKS)
{
    VkSparseMemoryBind* pBind = &binds[bindCount];
    pBind->resourceOffset = offset;

    AllocateOrGetMemoryRange(
        device,
        &memoryRequirements,
        &pBind->memory,
        &pBind->memoryOffset,
        &pBind->size);

    // memory ranges must be sized as multiples of the alignment
    assert(IsMultiple(pBind->size, memoryRequirements.alignment));
    assert(IsMultiple(pBind->memoryOffset, memoryRequirements.alignment));

    memoryRequirements.size -= pBind->size;
    offset                 += pBind->size;
    bindCount++;
}

// Ensure all image has backing
if (memoryRequirements.size)
{
    // Error condition - too many chunks
}

const VkSparseImageOpaqueMemoryBindInfo opaqueBindInfo =
{
    sparseImage,                                     // image
    bindCount,                                       // bindCount
    binds,                                           // pBinds
};

```

```

const VkBindSparseInfo bindSparseInfo =
{
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO,           // sType
    NULL,                                       // pNext
    ...
    1,                                           // imageOpaqueBindCount
    &opaqueBindInfo,                           // pImageOpaqueBinds
    ...
};

// vkQueueBindSparse is externally synchronized per queue object.
AcquireQueueOwnership(queue);

// Actually bind memory
vkQueueBindSparse(queue, 1, &bindSparseInfo, VK_NULL_HANDLE);

ReleaseQueueOwnership(queue);

```

### 31.8.2. Advanced Sparse Resources

This more advanced example creates an arrayed color attachment / texture image and binds only LOD zero and the **required** metadata to physical memory.

```

VkDevice                               device;
VkQueue                                queue;
VkImage                                 sparseImage;
VkAllocationCallbacks*                  pAllocator = NULL;
VkMemoryRequirements                   memoryRequirements = {};
VkSparseImageMemoryRequirements*       sparseRequirementsCount = 0;
VkSparseMemoryBind                     pSparseReqs = NULL;
VkSparseImageMemoryBind                binds[MY_IMAGE_ARRAY_SIZE] = {};
VkSparseImageMemoryBind                imageBinds[MY_IMAGE_ARRAY_SIZE] = {};
uint32_t                                bindCount = 0;

// Allocate image object (both renderable and sampleable)
const VkImageCreateInfo sparseImageInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,           // sType
    NULL,                                         // pNext
    VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT | ..., // flags
    ...
    VK_FORMAT_R8G8B8A8_UNORM,                    // format
    ...
    MY_IMAGE_ARRAY_SIZE,                         // arrayLayers
    ...
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT |        // usage
    VK_IMAGE_USAGE_SAMPLED_BIT,
    ...
};

vkCreateImage(device, &sparseImageInfo, pAllocator, &sparseImage);

```

```

// Get memory requirements
vkGetImageMemoryRequirements(
    device,
    sparseImage,
    &memoryRequirements);

// Get sparse image aspect properties
vkGetImageSparseMemoryRequirements(
    device,
    sparseImage,
    &sparseRequirementsCount,
    NULL);

pSparseReqs = (VkSparseImageMemoryRequirements*)
    malloc(sparseRequirementsCount * sizeof(VkSparseImageMemoryRequirements));

vkGetImageSparseMemoryRequirements(
    device,
    sparseImage,
    &sparseRequirementsCount,
    pSparseReqs);

// Bind LOD level 0 and any required metadata to memory
for (uint32_t i = 0; i < sparseRequirementsCount; ++i)
{
    if (pSparseReqs[i].formatProperties.aspectMask &
        VK_IMAGE_ASPECT_METADATA_BIT)
    {
        // Metadata must not be combined with other aspects
        assert(pSparseReqs[i].formatProperties.aspectMask ==
               VK_IMAGE_ASPECT_METADATA_BIT);

        if (pSparseReqs[i].formatProperties.flags &
            VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT)
        {
            VkSparseMemoryBind* pBind = &binds[bindCount];
            pBind->memorySize = pSparseReqs[i].imageMipTailSize;
            bindCount++;

            // ... Allocate memory range

            pBind->resourceOffset = pSparseReqs[i].imageMipTailOffset;
            pBind->memoryOffset = /* allocated memoryOffset */;
            pBind->memory = /* allocated memory */;
            pBind->flags = VK_SPARSE_MEMORY_BIND_METADATA_BIT;

        }
        else
        {
            // Need a mip tail region per array layer.
        }
    }
}

```

```

    for (uint32_t a = 0; a < sparseImageInfo.arrayLayers; ++a)
    {
        VkSparseMemoryBind* pBind = &binds[bindCount];
        pBind->memorySize = pSparseReqs[i].imageMipTailSize;
        bindCount++;

        // ... Allocate memory range

        pBind->resourceOffset = pSparseReqs[i].imageMipTailOffset +
            (a * pSparseReqs[i].imageMipTailStride);

        pBind->memoryOffset = /* allocated memoryOffset */;
        pBind->memory = /* allocated memory */
        pBind->flags = VK_SPARSE_MEMORY_BIND_METADATA_BIT;
    }
}

else
{
    // resource data
    VkExtent3D lod0BlockSize =
    {
        AlignedDivide(
            sparseImageInfo.extent.width,
            pSparseReqs[i].formatProperties.imageGranularity.width);
        AlignedDivide(
            sparseImageInfo.extent.height,
            pSparseReqs[i].formatProperties.imageGranularity.height);
        AlignedDivide(
            sparseImageInfo.extent.depth,
            pSparseReqs[i].formatProperties.imageGranularity.depth);
    }
    size_t totalBlocks =
        lod0BlockSize.width *
        lod0BlockSize.height *
        lod0BlockSize.depth;

    // Each block is the same size as the alignment requirement,
    // calculate total memory size for level 0
    VkDeviceSize lod0MemSize = totalBlocks * memoryRequirements.alignment;

    // Allocate memory for each array layer
    for (uint32_t a = 0; a < sparseImageInfo.arrayLayers; ++a)
    {
        // ... Allocate memory range

        VkSparseImageMemoryBind* pBind = &imageBinds[a];
        pBind->subresource.aspectMask = pSparseReqs[i].formatProperties.
aspectMask;
        pBind->subresource.mipLevel = 0;
        pBind->subresource.arrayLayer = a;
    }
}

```

```

    pBind->offset = (VkOffset3D){0, 0, 0};
    pBind->extent = sparseImageInfo.extent;
    pBind->memoryOffset = /* allocated memoryOffset */;
    pBind->memory = /* allocated memory */;
    pBind->flags = 0;
}
}

free(pSparseReqs);
}

const VkSparseImageOpaqueMemoryBindInfo opaqueBindInfo =
{
    sparseImage,                                // image
    bindCount,                                   // bindCount
    binds,                                       // pBinds
};

const VkSparseImageMemoryBindInfo imageBindInfo =
{
    sparseImage,                                // image
    sparseImageInfo.arrayLayers,                // bindCount
    imageBinds,                                  // pBinds
};

const VkBindSparseInfo bindSparseInfo =
{
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO,           // sType
    NULL,                                         // pNext
    ...
    1,                                            // imageOpaqueBindCount
    &opaqueBindInfo,                            // pImageOpaqueBinds
    1,                                            // imageBindCount
    &imageBindInfo,                            // pImageBinds
    ...
};

// vkQueueBindSparse is externally synchronized per queue object.
AcquireQueueOwnership(queue);

// Actually bind memory
vkQueueBindSparse(queue, 1, &bindSparseInfo, VK_NULL_HANDLE);

ReleaseQueueOwnership(queue);

```

# Chapter 32. Window System Integration (WSI)

This chapter discusses the window system integration (WSI) between the Vulkan API and the various forms of displaying the results of rendering to a user. Since the Vulkan API **can** be used without displaying results, WSI is provided through the use of optional Vulkan extensions. This chapter provides an overview of WSI. See the appendix for additional details of each WSI extension, including which extensions **must** be enabled in order to use each of the functions described in this chapter.

## 32.1. WSI Platform

A platform is an abstraction for a window system, OS, etc. Some examples include MS Windows, Android, and Wayland. The Vulkan API **may** be integrated in a unique manner for each platform.

The Vulkan API does not define any type of platform object. Platform-specific WSI extensions are defined, each containing platform-specific functions for using WSI. Use of these extensions is guarded by preprocessor symbols as defined in the [Window System-Specific Header Control](#) appendix.

In order for an application to be compiled to use WSI with a given platform, it must either:

- #define the appropriate preprocessor symbol prior to including the `vulkan.h` header file, or
- include `vulkan_core.h` and any native platform headers, followed by the appropriate platform-specific header.

The preprocessor symbols and platform-specific headers are defined in the [Window System Extensions and Headers](#) table.

Each platform-specific extension is an instance extension. The application **must** enable instance extensions with `vkCreateInstance` before using them.

## 32.2. WSI Surface

Native platform surface or window objects are abstracted by surface objects, which are represented by `VkSurfaceKHR` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSurfaceKHR)
```

The `VK_KHR_surface` extension declares the `VkSurfaceKHR` object, and provides a function for destroying `VkSurfaceKHR` objects. Separate platform-specific extensions each provide a function for creating a `VkSurfaceKHR` object for the respective platform. From the application's perspective this is an opaque handle, just like the handles of other Vulkan objects.

#### *Note*

On certain platforms, the Vulkan loader and ICDs **may** have conventions that treat the handle as a pointer to a structure containing the platform-specific information about the surface. This will be described in the documentation for the loader-ICD interface, and in the `vk_icd.h` header file of the LoaderAndTools source-code repository. This does not affect the loader-layer interface; layers **may** wrap `VkSurfaceKHR` objects.



### 32.2.1. Android Platform

To create a `VkSurfaceKHR` object for an Android native window, call:

```
VkResult vkCreateAndroidSurfaceKHR(  
    VkInstance instance,  
    const VkAndroidSurfaceCreateInfoKHR* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkAndroidSurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

During the lifetime of a surface created using a particular `ANativeWindow` handle any attempts to create another surface for the same `ANativeWindow` and any attempts to connect to the same `ANativeWindow` through other platform mechanisms will fail.

#### *Note*



In particular, only one `VkSurfaceKHR` **can** exist at a time for a given window. Similarly, a native window **cannot** be used by both a `VkSurfaceKHR` and `EGLSurface` simultaneously.

If successful, `vkCreateAndroidSurfaceKHR` increments the `ANativeWindow`'s reference count, and `vkDestroySurfaceKHR` will decrement it.

On Android, when a swapchain's `imageExtent` does not match the surface's `currentExtent`, the presentable images will be scaled to the surface's dimensions during presentation. `minImageExtent` is (1,1), and `maxImageExtent` is the maximum image size supported by the consumer. For the system compositor, `currentExtent` is the window size (i.e. the consumer's preferred size).

## Valid Usage (Implicit)

- **instance** **must** be a valid `VkInstance` handle
- **pCreateInfo** **must** be a valid pointer to a valid `VkAndroidSurfaceCreateInfoKHR` structure
- If **pAllocator** is not `NULL`, **pAllocator** **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **pSurface** **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkAndroidSurfaceCreateInfoKHR` structure is defined as:

```
typedef struct VkAndroidSurfaceCreateInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkAndroidSurfaceCreateFlagsKHR flags;
    struct ANativeWindow*     window;
} VkAndroidSurfaceCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **window** is a pointer to the `ANativeWindow` to associate the surface with.

## Valid Usage

- **window** **must** point to a valid Android `ANativeWindow`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ANDROID_SURFACE_CREATE_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

To remove an unnecessary compile-time dependency, an incomplete type definition of `ANativeWindow` is provided in the Vulkan headers:

```
struct ANativeWindow;
```

The actual `ANativeWindow` type is defined in Android NDK headers.

### 32.2.2. Wayland Platform

To create a `VkSurfaceKHR` object for a Wayland surface, call:

```
VkResult vkCreateWaylandSurfaceKHR(  
    VkInstance instance,  
    const VkWaylandSurfaceCreateInfoKHR* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkWaylandSurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkWaylandSurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkWaylandSurfaceCreateInfoKHR` structure is defined as:

```
typedef struct VkWaylandSurfaceCreateInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkWaylandSurfaceCreateFlagsKHR flags;
    struct wl_display*        display;
    struct wl_surface*        surface;
} VkWaylandSurfaceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `display` and `surface` are pointers to the Wayland `wl_display` and `wl_surface` to associate the surface with.

### Valid Usage

- `display` **must** point to a valid Wayland `wl_display`.
- `surface` **must** point to a valid Wayland `wl_surface`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WAYLAND_SURFACE_CREATE_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

On Wayland, `currentExtent` is the special value (0xFFFFFFFF, 0xFFFFFFFF), indicating that the surface size will be determined by the extent of a swapchain targeting the surface. Whatever the application sets a swapchain's `imageExtent` to will be the size of the window, after the first image is presented. `minImageExtent` is (1,1), and `maxImageExtent` is the maximum supported surface size. Any calls to `vkGetPhysicalDeviceSurfacePresentModesKHR` on a surface created with `vkCreateWaylandSurfaceKHR` are **required** to return `VK_PRESENT_MODE_MAILBOX_KHR` as one of the valid

present modes.

Some Vulkan functions **may** send protocol over the specified `wl_display` connection when using a swapchain or presentable images created from a `VkSurfaceKHR` referring to a `wl_surface`. Applications **must** therefore ensure that both the `wl_display` and the `wl_surface` remain valid for the lifetime of any `VkSwapchainKHR` objects created from a particular `wl_display` and `wl_surface`. Also, calling `vkQueuePresentKHR` will result in Vulkan sending `wl_surface.commit` requests to the underlying `wl_surface` of each `VkSwapchainKHR` objects referenced by `pPresentInfo`. If the swapchain is created with a present mode of `VK_PRESENT_MODE_MAILBOX_KHR` or `VK_PRESENT_MODE_IMMEDIATE_KHR`, then the corresponding `wl_surface.attach`, `wl_surface.damage`, and `wl_surface.commit` request **must** be issued by the implementation during the call to `vkQueuePresentKHR` and **must** not be issued by the implementation outside of `vkQueuePresentKHR`. This ensures that any Wayland requests sent by the client after the call to `vkQueuePresentKHR` returns will be received by the compositor after the `wl_surface.commit`. Regardless of the mode of swapchain creation, a new `wl_event_queue` **must** be created for each successful `vkCreateWaylandSurfaceKHR` call, and every Wayland object created by the implementation **must** be assigned to this event queue. If the platform provides Wayland 1.11 or greater, this **must** be implemented by the use of Wayland proxy object wrappers, to avoid race conditions.

If the application wishes to synchronize any window changes with a particular frame, such requests **must** be sent to the Wayland display server prior to calling `vkQueuePresentKHR`. For full control over interactions between Vulkan rendering and other Wayland protocol requests and events, a present mode of `VK_PRESENT_MODE_MAILBOX_KHR` **should** be used.

### 32.2.3. Win32 Platform

To create a `VkSurfaceKHR` object for a Win32 window, call:

```
VkResult vkCreateWin32SurfaceKHR(  
    VkInstance                                     instance,  
    const VkWin32SurfaceCreateInfoKHR* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkWin32SurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

## Valid Usage (Implicit)

- **instance** **must** be a valid `VkInstance` handle
- **pCreateInfo** **must** be a valid pointer to a valid `VkWin32SurfaceCreateInfoKHR` structure
- If **pAllocator** is not `NULL`, **pAllocator** **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **pSurface** **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkWin32SurfaceCreateInfoKHR` structure is defined as:

```
typedef struct VkWin32SurfaceCreateInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkWin32SurfaceCreateFlagsKHR flags;
    HINSTANCE                 hinstance;
    HWND                      hwnd;
} VkWin32SurfaceCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **hinstance** is the Win32 `HINSTANCE` for the window to associate the surface with.
- **hwnd** is the Win32 `HWND` for the window to associate the surface with.

## Valid Usage

- **hinstance** **must** be a valid Win32 `HINSTANCE`.
- **hwnd** **must** be a valid Win32 `HWND`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

With Win32, `minImageExtent`, `maxImageExtent`, and `currentExtent` **must** always equal the window size.

The `currentExtent` of a Win32 surface **must** have both `width` and `height` greater than 0, or both of them 0.

*Note*



Due to above restrictions, it is only possible to create a new swapchain on this platform with `imageExtent` being equal to the current size of the window.

The window size **may** become `(0, 0)` on this platform (e.g. when the window is minimized), and so a swapchain **cannot** be created until the size changes.

```
typedef VkFlags VkWin32SurfaceCreateFlagsKHR;
```

`VkWin32SurfaceCreateFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.

### 32.2.4. XCB Platform

To create a `VkSurfaceKHR` object for an X11 window, using the XCB client-side library, call:

```
VkResult vkCreateXcbSurfaceKHR(  
    VkInstance instance,  
    const VkXcbSurfaceCreateInfoKHR* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkXcbSurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkXcbSurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkXcbSurfaceCreateInfoKHR` structure is defined as:

```
typedef struct VkXcbSurfaceCreateInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkXcbSurfaceCreateFlagsKHR flags;
    xcb_connection_t*         connection;
    xcb_window_t              window;
} VkXcbSurfaceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `connection` is a pointer to an `xcb_connection_t` to the X server.
- `window` is the `xcb_window_t` for the X11 window to associate the surface with.

## Valid Usage

- `connection` **must** point to a valid X11 `xcb_connection_t`.
- `window` **must** be a valid X11 `xcb_window_t`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

With Xcb, `minImageExtent`, `maxImageExtent`, and `currentExtent` **must** always equal the window size.

The `currentExtent` of an Xcb surface **must** have both `width` and `height` greater than 0, or both of them 0.

*Note*



Due to above restrictions, it is only possible to create a new swapchain on this platform with `imageExtent` being equal to the current size of the window.

The window size **may** become `(0, 0)` on this platform (e.g. when the window is minimized), and so a swapchain **cannot** be created until the size changes.

Some Vulkan functions **may** send protocol over the specified xcb connection when using a swapchain or presentable images created from a `VkSurfaceKHR` referring to an xcb window. Applications **must** therefore ensure the xcb connection is available to Vulkan for the duration of any functions that manipulate such swapchains or their presentable images, and any functions that build or queue command buffers that operate on such presentable images. Specifically, applications using Vulkan with xcb-based swapchains **must**

- Avoid holding a server grab on an xcb connection while waiting for Vulkan operations to complete using a swapchain derived from a different xcb connection referring to the same X server instance. Failing to do so **may** result in deadlock.

### 32.2.5. Xlib Platform

To create a `VkSurfaceKHR` object for an X11 window, using the Xlib client-side library, call:

```
VkResult vkCreateXlibSurfaceKHR(  
    VkInstance instance,  
    const VkXlibSurfaceCreateInfoKHR* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkXlibSurfaceCreateInfoKHR` structure containing the parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

## Valid Usage (Implicit)

- **instance** **must** be a valid `VkInstance` handle
- **pCreateInfo** **must** be a valid pointer to a valid `VkXlibSurfaceCreateInfoKHR` structure
- If **pAllocator** is not `NULL`, **pAllocator** **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **pSurface** **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkXlibSurfaceCreateInfoKHR` structure is defined as:

```
typedef struct VkXlibSurfaceCreateInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkXlibSurfaceCreateFlagsKHR flags;
    Display*                 dpy;
    Window                   window;
} VkXlibSurfaceCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **dpy** is a pointer to an Xlib `Display` connection to the X server.
- **window** is an Xlib `Window` to associate the surface with.

## Valid Usage

- **dpy** **must** point to a valid Xlib `Display`.
- **window** **must** be a valid Xlib `Window`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR`
- `pNext` must be `NULL`
- `flags` must be `0`

With Xlib, `minImageExtent`, `maxImageExtent`, and `currentExtent` must always equal the window size.

The `currentExtent` of an Xlib surface must have both `width` and `height` greater than 0, or both of them 0.

*Note*



Due to above restrictions, it is only possible to create a new swapchain on this platform with `imageExtent` being equal to the current size of the window.

The window size may become `(0, 0)` on this platform (e.g. when the window is minimized), and so a swapchain cannot be created until the size changes.

Some Vulkan functions may send protocol over the specified Xlib `Display` connection when using a swapchain or presentable images created from a `VkSurfaceKHR` referring to an Xlib window. Applications must therefore ensure the display connection is available to Vulkan for the duration of any functions that manipulate such swapchains or their presentable images, and any functions that build or queue command buffers that operate on such presentable images. Specifically, applications using Vulkan with Xlib-based swapchains must

- Avoid holding a server grab on a display connection while waiting for Vulkan operations to complete using a swapchain derived from a different display connection referring to the same X server instance. Failing to do so may result in deadlock.

Some implementations may require threads to implement some presentation modes so applications must call `XInitThreads()` before calling any other Xlib functions.

### 32.2.6. Fuchsia Platform

To create a `VkSurfaceKHR` object for a Fuchsia ImagePipe, call:

```
VkResult vkCreateImagePipeSurfaceFUCHSIA(  
    VkInstance                           instance,  
    const VkImagePipeSurfaceCreateInfoFUCHSIA* pCreateInfo,  
    const VkAllocationCallbacks*           pAllocator,  
    VkSurfaceKHR*                      pSurface);
```

- `instance` is the instance to associate with the surface.
- `pCreateInfo` is a pointer to a `VkImagePipeSurfaceCreateInfoFUCHSIA` structure containing parameters affecting the creation of the surface object.

- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkImagePipeSurfaceCreateInfoFUCHSIA` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkImagePipeSurfaceCreateInfoFUCHSIA` structure is defined as:

```
typedef struct VkImagePipeSurfaceCreateInfoFUCHSIA {
    VkStructureType           sType;
    const void*                pNext;
    VkImagePipeSurfaceCreateFlagsFUCHSIA flags;
    zx_handle_t                  imagePipeHandle;
} VkImagePipeSurfaceCreateInfoFUCHSIA;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `imagePipeHandle` is a `zx_handle_t` referring to the ImagePipe to associate with the surface.

## Valid Usage

- `imagePipeHandle` **must** be a valid `zx_handle_t`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGEPIPE_SURFACE_CREATE_INFO_FUCHSIA`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

On Fuchsia, the surface `currentExtent` is the special value (0xFFFFFFFF, 0xFFFFFFFF), indicating that the surface size will be determined by the extent of a swapchain targeting the surface.

### 32.2.7. Google Games Platform

To create a `VkSurfaceKHR` object for a Google Games Platform stream descriptor, call:

```
VkResult vkCreateStreamDescriptorSurfaceGGP(  
    VkInstance                      instance,  
    const VkStreamDescriptorSurfaceCreateInfoGGP* pCreateInfo,  
    const VkAllocationCallbacks*      pAllocator,  
    VkSurfaceKHR*                   pSurface);
```

- `instance` is the instance to associate with the surface.
- `pCreateInfo` is a pointer to a `VkStreamDescriptorSurfaceCreateInfoGGP` structure containing parameters that affect the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkStreamDescriptorSurfaceCreateInfoGGP` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkStreamDescriptorSurfaceCreateInfoGGP` structure is defined as:

```
typedef struct VkStreamDescriptorSurfaceCreateInfoGGP {
    VkStructureType           sType;
    const void*               pNext;
    VkStreamDescriptorSurfaceCreateFlagsGGP   flags;
    GgpStreamDescriptor        streamDescriptor;
} VkStreamDescriptorSurfaceCreateInfoGGP;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `streamDescriptor` is a `GgpStreamDescriptor` referring to the GGP stream descriptor to associate with the surface.

### Valid Usage

- `streamDescriptor` **must** be a valid `GgpStreamDescriptor`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_STREAM_DESCRIPTOR_SURFACE_CREATE_INFO_GGP`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

On Google Games Platform, the surface extents are dynamic. The `minImageExtent` will never be greater than 1080p and the `maxImageExtent` will never be less than 1080p. The `currentExtent` will reflect the current optimal resolution.

Applications are expected to choose an appropriate size for the swapchain's `imageExtent`, within the bounds of the surface. Using the surface's `currentExtent` will offer the best performance and quality. When a swapchain's `imageExtent` does not match the surface's `currentExtent`, the presentable

images are scaled to the surface's dimensions during presentation if possible and `VK_SUBOPTIMAL_KHR` is returned, otherwise presentation fails with `VK_ERROR_OUT_OF_DATE_KHR`.

### 32.2.8. iOS Platform

To create a `VkSurfaceKHR` object for an iOS `UIView`, call:

```
VkResult vkCreateIOSSurfaceMVK(  
    VkInstance instance,  
    const VkIOSSurfaceCreateInfoMVK* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance with which to associate the surface.
- `pCreateInfo` is a pointer to a `VkIOSSurfaceCreateInfoMVK` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

#### Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkIOSSurfaceCreateInfoMVK` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

#### Return Codes

##### Success

- `VK_SUCCESS`

##### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkIOSSurfaceCreateInfoMVK` structure is defined as:

```

typedef struct VkIOSSurfaceCreateInfoMVK {
    VkStructureType           sType;
    const void*              pNext;
    VkIOSSurfaceCreateFlagsMVK flags;
    const void*              pView;
} VkIOSSurfaceCreateInfoMVK;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **pView** is a reference to a **UIView** object which will display this surface. This **UIView** **must** be backed by a **CALayer** instance of type **CAMetalLayer**.

### Valid Usage

- **pView** **must** be a valid **UIView** and **must** be backed by a **CALayer** instance of type **CAMetalLayer**.

### Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_IOS\_SURFACE\_CREATE\_INFO\_MVK**
- **pNext** **must** be **NULL**
- **flags** **must** be **0**

## 32.2.9. macOS Platform

To create a **VkSurfaceKHR** object for a macOS **NSView**, call:

```

VkResult vkCreateMacOSSurfaceMVK(
    VkInstance           instance,
    const VkMacOSSurfaceCreateInfoMVK* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR*       pSurface);

```

- **instance** is the instance with which to associate the surface.
- **pCreateInfo** is a pointer to a **VkMacOSSurfaceCreateInfoMVK** structure containing parameters affecting the creation of the surface object.
- **pAllocator** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- **pSurface** is a pointer to a **VkSurfaceKHR** handle in which the created surface object is returned.

## Valid Usage (Implicit)

- **instance** **must** be a valid `VkInstance` handle
- **pCreateInfo** **must** be a valid pointer to a valid `VkMacOSSurfaceCreateInfoMVK` structure
- If **pAllocator** is not `NULL`, **pAllocator** **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **pSurface** **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkMacOSSurfaceCreateInfoMVK` structure is defined as:

```
typedef struct VkMacOSSurfaceCreateInfoMVK {
    VkStructureType          sType;
    const void*              pNext;
    VkMacOSSurfaceCreateFlagsMVK flags;
    const void*              pView;
} VkMacOSSurfaceCreateInfoMVK;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **pView** is a reference to a `NSView` object which will display this surface. This `NSView` **must** be backed by a `CALayer` instance of type `CAMetalLayer`.

## Valid Usage

- **pView** **must** be a valid `NSView` and **must** be backed by a `CALayer` instance of type `CAMetalLayer`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

### 32.2.10. VI Platform

To create a `VkSurfaceKHR` object for an `nn::vi::Layer`, query the layer's native handle using `nn::vi::GetNativeWindow`, and then call:

```
VkResult vkCreateViSurfaceNN(  
    VkInstance instance,  
    const VkViSurfaceCreateInfoNN* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance with which to associate the surface.
- `pCreateInfo` is a pointer to a `VkViSurfaceCreateInfoNN` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

During the lifetime of a surface created using a particular `nn::vi::NativeWindowHandle`, applications **must** not attempt to create another surface for the same `nn::vi::Layer` or attempt to connect to the same `nn::vi::Layer` through other platform mechanisms.

If the native window is created with a specified size, `currentExtent` will reflect that size. In this case, applications should use the same size for the swapchain's `imageExtent`. Otherwise, the `currentExtent` will have the special value (0xFFFFFFFF, 0xFFFFFFFF), indicating that applications are expected to choose an appropriate size for the swapchain's `imageExtent` (e.g., by matching the result of a call to `nn::vi::GetDisplayResolution`).

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkViSurfaceCreateInfoNN` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkViSurfaceCreateInfoNN` structure is defined as:

```
typedef struct VkViSurfaceCreateInfoNN {
    VkStructureType          sType;
    const void*              pNext;
    VkViSurfaceCreateFlagsNN flags;
    void*                    window;
} VkViSurfaceCreateInfoNN;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `window` is the `nn::vi::NativeWindowHandle` for the `nn::vi::Layer` with which to associate the surface.

### Valid Usage

- `window` **must** be a valid `nn::vi::NativeWindowHandle`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_VI_SURFACE_CREATE_INFO_NN`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

## 32.2.11. Metal Platform

To create a `VkSurfaceKHR` object for a `CAMetalLayer`, call:

```

VkResult vkCreateMetalSurfaceEXT(
    VkInstance instance,
    const VkMetalSurfaceCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);

```

- `instance` is the instance with which to associate the surface.
- `pCreateInfo` is a pointer to a `VkMetalSurfaceCreateInfoEXT` structure specifying parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkMetalSurfaceCreateInfoEXT` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkMetalSurfaceCreateInfoEXT` structure is defined as:

```

typedef struct VkMetalSurfaceCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkMetalSurfaceCreateFlagsEXT flags;
    const CAMetalLayer* pLayer;
} VkMetalSurfaceCreateInfoEXT;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `flags` is reserved for future use.
- `pLayer` is a reference to a [CAMetalLayer](#) object representing a renderable surface.

## Valid Usage

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_METAL_SURFACE_CREATE_INFO_EXT`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

To remove an unnecessary compile-time dependency, an incomplete type definition of [CAMetalLayer](#) is provided in the Vulkan headers:

```
#ifdef __OBJC__
@class CAMetalLayer;
#else
typedef void CAMetalLayer;
#endif
```

The actual [CAMetalLayer](#) type is defined in the QuartzCore framework.

### 32.2.12. Platform-Independent Information

Once created, [VkSurfaceKHR](#) objects **can** be used in this and other extensions, in particular the [VK\\_KHR\\_swapchain](#) extension.

Several WSI functions return [VK\\_ERROR\\_SURFACE\\_LOST\\_KHR](#) if the surface becomes no longer available. After such an error, the surface (and any child swapchain, if one exists) **should** be destroyed, as there is no way to restore them to a not-lost state. Applications **may** attempt to create a new [VkSurfaceKHR](#) using the same native platform window object, but whether such re-creation will succeed is platform-dependent and **may** depend on the reason the surface became unavailable. A lost surface does not otherwise cause devices to be [lost](#).

To destroy a [VkSurfaceKHR](#) object, call:

```
void vkDestroySurfaceKHR(
    VkInstance                                     instance,
    VkSurfaceKHR                                    surface,
    const VkAllocationCallbacks* pAllocator);
```

- `instance` is the instance used to create the surface.
- `surface` is the surface to destroy.

- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).

Destroying a `VkSurfaceKHR` merely severs the connection between Vulkan and the native surface, and does not imply destroying the native surface, closing a window, or similar behavior.

## Valid Usage

- All `VkSwapchainKHR` objects created for `surface` **must** have been destroyed prior to destroying `surface`
- If `VkAllocationCallbacks` were provided when `surface` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `surface` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- If `surface` is not `VK_NULL_HANDLE`, `surface` **must** be a valid `VkSurfaceKHR` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `surface` is a valid handle, it **must** have been created, allocated, or retrieved from `instance`

## Host Synchronization

- Host access to `surface` **must** be externally synchronized

## 32.3. Presenting Directly to Display Devices

In some environments applications **can** also present Vulkan rendering directly to display devices without using an intermediate windowing system. This **can** be useful for embedded applications, or implementing the rendering/presentation backend of a windowing system using Vulkan. The `VK_KHR_display` extension provides the functionality necessary to enumerate display devices and create `VkSurfaceKHR` objects that target displays.

### 32.3.1. Display Enumeration

Displays are represented by `VkDisplayKHR` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDisplayKHR)
```

Various functions are provided for enumerating the available display devices present on a Vulkan physical device. To query information about the available displays, call:

```
VkResult vkGetPhysicalDeviceDisplayPropertiesKHR(  
    VkPhysicalDevice physicalDevice,  
    uint32_t* pPropertyCount,  
    VkDisplayPropertiesKHR* pProperties);
```

- `physicalDevice` is a physical device.
- `pPropertyCount` is a pointer to an integer related to the number of display devices available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayPropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of display devices available for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display devices for `physicalDevice`, at most `pPropertyCount` structures will be written. If `pPropertyCount` is smaller than the number of display devices available for `physicalDevice`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkDisplayPropertiesKHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPropertiesKHR` structure is defined as:

```

typedef struct VkDisplayPropertiesKHR {
    VkDisplayKHR display;
    const char* displayName;
    VkExtent2D physicalDimensions;
    VkExtent2D physicalResolution;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkBool32 planeReorderPossible;
    VkBool32 persistentContent;
} VkDisplayPropertiesKHR;

```

- **display** is a handle that is used to refer to the display described here. This handle will be valid for the lifetime of the Vulkan instance.
- **displayName** is a pointer to a null-terminated UTF-8 string containing the name of the display. Generally, this will be the name provided by the display's EDID. It **can** be **NULL** if no suitable name is available. If not **NULL**, the memory it points to **must** remain accessible as long as **display** is valid.
- **physicalDimensions** describes the physical width and height of the visible portion of the display, in millimeters.
- **physicalResolution** describes the physical, native, or preferred resolution of the display.

*Note*



For devices which have no natural value to return here, implementations **should** return the maximum resolution supported.

- **supportedTransforms** is a bitmask of [VkSurfaceTransformFlagBitsKHR](#) describing which transforms are supported by this display.
- **planeReorderPossible** tells whether the planes on this display **can** have their z order changed. If this is **VK\_TRUE**, the application **can** re-arrange the planes on this display in any order relative to each other.
- **persistentContent** tells whether the display supports self-refresh/internal buffering. If this is true, the application **can** submit persistent present operations on swapchains created against this display.

*Note*



Persistent presents **may** have higher latency, and **may** use less power when the screen content is updated infrequently, or when only a portion of the screen needs to be updated in most frames.

To query information about the available displays, call:

```

VkResult vkGetPhysicalDeviceDisplayProperties2KHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkDisplayProperties2KHR* pProperties);

```

- `physicalDevice` is a physical device.
- `pPropertyCount` is a pointer to an integer related to the number of display devices available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayProperties2KHR` structures.

`vkGetPhysicalDeviceDisplayProperties2KHR` behaves similarly to `vkGetPhysicalDeviceDisplayPropertiesKHR`, with the ability to return extended information via chained output structures.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkDisplayProperties2KHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayProperties2KHR` structure is defined as:

```
typedef struct VkDisplayProperties2KHR {
    VkStructureType          sType;
    void*                    pNext;
    VkDisplayPropertiesKHR   displayProperties;
} VkDisplayProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `displayProperties` is an instance of the `VkDisplayPropertiesKHR` structure.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR`
- `pNext` **must** be `NULL`

## Acquiring and Releasing Displays

On some platforms, access to displays is limited to a single process or native driver instance. On such platforms, some or all of the displays may not be available to Vulkan if they are already in use by a native windowing system or other application.

To acquire permission to directly access a display in Vulkan from an X11 server, call:

```
VkResult vkAcquireXlibDisplayEXT(  
    VkPhysicalDevice physicalDevice,  
    Display* dpy,  
    VkDisplayKHR display);
```

- **physicalDevice** The physical device the display is on.
- **dpy** A connection to the X11 server that currently owns **display**.
- **display** The display the caller wishes to control in Vulkan.

All permissions necessary to control the display are granted to the Vulkan instance associated with **physicalDevice** until the display is released or the X11 connection specified by **dpy** is terminated. Permission to access the display **may** be temporarily revoked during periods when the X11 server from which control was acquired itself loses access to **display**. During such periods, operations which require access to the display **must** fail with an appropriate error code. If the X11 server associated with **dpy** does not own **display**, or if permission to access it has already been acquired by another entity, the call **must** return the error code **VK\_ERROR\_INITIALIZATION\_FAILED**.

*Note*



One example of when an X11 server loses access to a display is when it loses ownership of its virtual terminal.

### Valid Usage (Implicit)

- **physicalDevice** **must** be a valid **VkPhysicalDevice** handle
- **dpy** **must** be a valid pointer to a **Display** value
- **display** **must** be a valid **VkDisplayKHR** handle

### Return Codes

#### Success

- **VK\_SUCCESS**

#### Failure

- **VK\_ERROR\_INITIALIZATION\_FAILED**

When acquiring displays from an X11 server, an application may also wish to enumerate and

identify them using a native handle rather than a `VkDisplayKHR` handle. To determine the `VkDisplayKHR` handle corresponding to an X11 RandR Output, call:

```
VkResult vkGetRandROutputDisplayEXT(  
    VkPhysicalDevice physicalDevice,  
    Display* dpy,  
    RROutput rrOutput,  
    VkDisplayKHR* pDisplay);
```

- `physicalDevice` The physical device to query the display handle on.
- `dpy` A connection to the X11 server from which `rrOutput` was queried.
- `rrOutput` An X11 RandR output ID.
- `pDisplay` The corresponding `VkDisplayKHR` handle will be returned here.

If there is no `VkDisplayKHR` corresponding to `rrOutput` on `physicalDevice`, `VK_NULL_HANDLE` must be returned in `pDisplay`.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `dpy` **must** be a valid pointer to a `Display` value
- `pDisplay` **must** be a valid pointer to a `VkDisplayKHR` handle

### Return Codes

#### Success

- `VK_SUCCESS`

To release a previously acquired display, call:

```
VkResult vkReleaseDisplayEXT(  
    VkPhysicalDevice physicalDevice,  
    VkDisplayKHR display);
```

- `physicalDevice` The physical device the display is on.
- `display` The display to release control of.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `display` **must** be a valid `VkDisplayKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

## Display Planes

Images are presented to individual planes on a display. Devices **must** support at least one plane on each display. Planes **can** be stacked and blended to composite multiple images on one display. Devices **may** support only a fixed stacking order and fixed mapping between planes and displays, or they **may** allow arbitrary application specified stacking orders and mappings between planes and displays. To query the properties of device display planes, call:

```
VkResult vkGetPhysicalDeviceDisplayPlanePropertiesKHR(  
    VkPhysicalDevice                      physicalDevice,  
    uint32_t*                            pPropertyCount,  
    VkDisplayPlanePropertiesKHR*          pProperties);
```

- `physicalDevice` is a physical device.
- `pPropertyCount` is a pointer to an integer related to the number of display planes available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayPlanePropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of display planes available for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display planes for `physicalDevice`, at most `pPropertyCount` structures will be written.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkDisplayPlanePropertiesKHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPlanePropertiesKHR` structure is defined as:

```
typedef struct VkDisplayPlanePropertiesKHR {  
    VkDisplayKHR    currentDisplay;  
    uint32_t        currentStackIndex;  
} VkDisplayPlanePropertiesKHR;
```

- `currentDisplay` is the handle of the display the plane is currently associated with. If the plane is not currently attached to any displays, this will be `VK_NULL_HANDLE`.
- `currentStackIndex` is the current z-order of the plane. This will be between 0 and the value returned by `vkGetPhysicalDeviceDisplayPlanePropertiesKHR` in `pPropertyCount`.

To query the properties of a device's display planes, call:

```
VkResult vkGetPhysicalDeviceDisplayPlaneProperties2KHR(  
    VkPhysicalDevice                      physicalDevice,  
    uint32_t*                            pPropertyCount,  
    VkDisplayPlaneProperties2KHR*         pProperties);
```

- `physicalDevice` is a physical device.
- `pPropertyCount` is a pointer to an integer related to the number of display planes available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayPlaneProperties2KHR` structures.

`vkGetPhysicalDeviceDisplayPlaneProperties2KHR` behaves similarly to `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`, with the ability to return extended information via chained output structures.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkDisplayPlaneProperties2KHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPlaneProperties2KHR` structure is defined as:

```
typedef struct VkDisplayPlaneProperties2KHR {
    VkStructureType           sType;
    void*                     pNext;
    VkDisplayPlanePropertiesKHR displayPlaneProperties;
} VkDisplayPlaneProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `displayPlaneProperties` is an instance of the `VkDisplayPlanePropertiesKHR` structure.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR`
- `pNext` **must** be `NULL`

To determine which displays a plane is usable with, call

```

VkResult vkGetDisplayPlaneSupportedDisplaysKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t planeIndex,
    uint32_t* pDisplayCount,
    VkDisplayKHR** pDisplays);

```

- `physicalDevice` is a physical device.
- `planeIndex` is the plane which the application wishes to use, and **must** be in the range [0, physical device plane count - 1].
- `pDisplayCount` is a pointer to an integer related to the number of displays available or queried, as described below.
- `pDisplays` is either `NULL` or a pointer to an array of `VkDisplayKHR` handles.

If `pDisplays` is `NULL`, then the number of displays usable with the specified `planeIndex` for `physicalDevice` is returned in `pDisplayCount`. Otherwise, `pDisplayCount` **must** point to a variable set by the user to the number of elements in the `pDisplays` array, and on return the variable is overwritten with the number of handles actually written to `pDisplays`. If the value of `pDisplayCount` is less than the number of display planes for `physicalDevice`, at most `pDisplayCount` handles will be written. If `pDisplayCount` is smaller than the number of displays usable with the specified `planeIndex` for `physicalDevice`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage

- `planeIndex` **must** be less than the number of display planes supported by the device as determined by calling `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pDisplayCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pDisplayCount` is not `0`, and `pDisplays` is not `NULL`, `pDisplays` **must** be a valid pointer to an array of `pDisplayCount` `VkDisplayKHR` handles

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Additional properties of displays are queried using specialized query functions.

## Display Modes

Display modes are represented by `VkDisplayModeKHR` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDisplayModeKHR)
```

Each display has one or more supported modes associated with it by default. These built-in modes are queried by calling:

```
VkResult vkGetDisplayModePropertiesKHR(  
    VkPhysicalDevice           physicalDevice,  
    VkDisplayKHR               display,  
    uint32_t*                  pPropertyCount,  
    VkDisplayModePropertiesKHR* pProperties);
```

- `physicalDevice` is the physical device associated with `display`.
- `display` is the display to query.
- `pPropertyCount` is a pointer to an integer related to the number of display modes available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayModePropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of display modes available on the specified `display` for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display modes for `physicalDevice`, at most `pPropertyCount` structures will be written. If `pPropertyCount` is smaller than the number of display modes available on the specified `display` for `physicalDevice`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `display` **must** be a valid `VkDisplayKHR` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkDisplayModePropertiesKHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayModePropertiesKHR` structure is defined as:

```
typedef struct VkDisplayModePropertiesKHR {
    VkDisplayModeKHR             displayMode;
    VkDisplayModeParametersKHR   parameters;
} VkDisplayModePropertiesKHR;
```

- `displayMode` is a handle to the display mode described in this structure. This handle will be valid for the lifetime of the Vulkan instance.
- `parameters` is a `VkDisplayModeParametersKHR` structure describing the display parameters associated with `displayMode`.

```
typedef VkFlags VkDisplayModeCreateFlagsKHR;
```

`VkDisplayModeCreateFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.

To query the properties of a device's built-in display modes, call:

```

VkResult vkGetDisplayModeProperties2KHR(
    VkPhysicalDevice physicalDevice,
    VkDisplayKHR display,
    uint32_t* pPropertyCount,
    VkDisplayModeProperties2KHR* pProperties);

```

- `physicalDevice` is the physical device associated with `display`.
- `display` is the display to query.
- `pPropertyCount` is a pointer to an integer related to the number of display modes available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayModeProperties2KHR` structures.

`vkGetDisplayModeProperties2KHR` behaves similarly to `vkGetDisplayModePropertiesKHR`, with the ability to return extended information via chained output structures.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `display` **must** be a valid `VkDisplayKHR` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkDisplayModeProperties2KHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayModeProperties2KHR` structure is defined as:

```

typedef struct VkDisplayModeProperties2KHR {
    VkStructureType sType;
    void* pNext;
    VkDisplayModePropertiesKHR displayModeProperties;
} VkDisplayModeProperties2KHR;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `displayModeProperties` is an instance of the `VkDisplayModePropertiesKHR` structure.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR`
- `pNext` **must** be `NULL`

The `VkDisplayModeParametersKHR` structure is defined as:

```
typedef struct VkDisplayModeParametersKHR {
    VkExtent2D    visibleRegion;
    uint32_t      refreshRate;
} VkDisplayModeParametersKHR;
```

- `visibleRegion` is the 2D extents of the visible region.
- `refreshRate` is a `uint32_t` that is the number of times the display is refreshed each second multiplied by 1000.



*Note*

For example, a 60Hz display mode would report a `refreshRate` of 60,000.

### Valid Usage

- The `width` member of `visibleRegion` **must** be greater than `0`
- The `height` member of `visibleRegion` **must** be greater than `0`
- `refreshRate` **must** be greater than `0`

Additional modes **may** also be created by calling:

```
VkResult vkCreateDisplayModeKHR(
    VkPhysicalDevice                      physicalDevice,
    VkDisplayKHR                          display,
    const VkDisplayModeCreateInfoKHR*     pCreateInfo,
    const VkAllocationCallbacks*          pAllocator,
    VkDisplayModeKHR*                    pMode);
```

- `physicalDevice` is the physical device associated with `display`.
- `display` is the display to create an additional mode for.
- `pCreateInfo` is a `VkDisplayModeCreateInfoKHR` structure describing the new mode to create.

- `pAllocator` is the allocator used for host memory allocated for the display mode object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pMode` returns the handle of the mode created.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `display` **must** be a valid `VkDisplayKHR` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkDisplayModeCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pMode` **must** be a valid pointer to a `VkDisplayModeKHR` handle

## Host Synchronization

- Host access to `display` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`

The `VkDisplayModeCreateInfoKHR` structure is defined as:

```
typedef struct VkDisplayModeCreateInfoKHR {
    VkStructureType          sType;
    const void*               pNext;
    VkDisplayModeCreateFlagsKHR flags;
    VkDisplayModeParametersKHR parameters;
} VkDisplayModeCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use, and **must** be zero.
- `parameters` is a `VkDisplayModeParametersKHR` structure describing the display parameters to use in creating the new mode. If the parameters are not compatible with the specified display,

the implementation **must** return `VK_ERROR_INITIALIZATION_FAILED`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `parameters` **must** be a valid `VkDisplayModeParametersKHR` structure

Applications that wish to present directly to a display **must** select which layer, or “plane” of the display they wish to target, and a mode to use with the display. Each display supports at least one plane. The capabilities of a given mode and plane combination are determined by calling:

```
VkResult vkGetDisplayPlaneCapabilitiesKHR(  
    VkPhysicalDevice           physicalDevice,  
    VkDisplayModeKHR          mode,  
    uint32_t                   planeIndex,  
    VkDisplayPlaneCapabilitiesKHR* pCapabilities);
```

- `physicalDevice` is the physical device associated with `display`
- `mode` is the display mode the application intends to program when using the specified plane. Note this parameter also implicitly specifies a display.
- `planeIndex` is the plane which the application intends to use with the display, and is less than the number of display planes supported by the device.
- `pCapabilities` is a pointer to a `VkDisplayPlaneCapabilitiesKHR` structure in which the capabilities are returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `mode` **must** be a valid `VkDisplayModeKHR` handle
- `pCapabilities` **must** be a valid pointer to a `VkDisplayPlaneCapabilitiesKHR` structure

## Host Synchronization

- Host access to `mode` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPlaneCapabilitiesKHR` structure is defined as:

```
typedef struct VkDisplayPlaneCapabilitiesKHR {
    VkDisplayPlaneAlphaFlagsKHR    supportedAlpha;
    VkOffset2D                    minSrcPosition;
    VkOffset2D                    maxSrcPosition;
    VkExtent2D                   minSrcExtent;
    VkExtent2D                   maxSrcExtent;
    VkOffset2D                    minDstPosition;
    VkOffset2D                    maxDstPosition;
    VkExtent2D                   minDstExtent;
    VkExtent2D                   maxDstExtent;
} VkDisplayPlaneCapabilitiesKHR;
```

- `supportedAlpha` is a bitmask of `VkDisplayPlaneAlphaFlagBitsKHR` describing the supported alpha blending modes.
- `minSrcPosition` is the minimum source rectangle offset supported by this plane using the specified mode.
- `maxSrcPosition` is the maximum source rectangle offset supported by this plane using the specified mode. The `x` and `y` components of `maxSrcPosition` **must** each be greater than or equal to the `x` and `y` components of `minSrcPosition`, respectively.
- `minSrcExtent` is the minimum source rectangle size supported by this plane using the specified mode.
- `maxSrcExtent` is the maximum source rectangle size supported by this plane using the specified mode.
- `minDstPosition`, `maxDstPosition`, `minDstExtent`, `maxDstExtent` all have similar semantics to their corresponding `*Src*` equivalents, but apply to the output region within the mode rather than the input region within the source image. Unlike the `*Src*` offsets, `minDstPosition` and `maxDstPosition` **may** contain negative values.

The minimum and maximum position and extent fields describe the implementation limits, if any, as they apply to the specified display mode and plane. Vendors **may** support displaying a subset of a swapchain's presentable images on the specified display plane. This is expressed by returning `minSrcPosition`, `maxSrcPosition`, `minSrcExtent`, and `maxSrcExtent` values that indicate a range of possible positions and sizes **may** be used to specify the region within the presentable images that source pixels will be read from when creating a swapchain on the specified display mode and

plane.

Vendors **may** also support mapping the presentable images' content to a subset or superset of the visible region in the specified display mode. This is expressed by returning `minDstPosition`, `maxDstPosition`, `minDstExtent` and `maxDstExtent` values that indicate a range of possible positions and sizes **may** be used to describe the region within the display mode that the source pixels will be mapped to.

Other vendors **may** support only a 1-1 mapping between pixels in the presentable images and the display mode. This **may** be indicated by returning (0,0) for `minSrcPosition`, `maxSrcPosition`, `minDstPosition`, and `maxDstPosition`, and (display mode width, display mode height) for `minSrcExtent`, `maxSrcExtent`, `minDstExtent`, and `maxDstExtent`.

These values indicate the limits of the implementation's individual fields. Not all combinations of values within the offset and extent ranges returned in `VkDisplayPlaneCapabilitiesKHR` are guaranteed to be supported. Presentation requests specifying unsupported combinations **may** fail.

To query the capabilities of a given mode and plane combination, call:

```
VkResult vkGetDisplayPlaneCapabilities2KHR(  
    VkPhysicalDevice physicalDevice,  
    const VkDisplayPlaneInfo2KHR* pDisplayPlaneInfo,  
    VkDisplayPlaneCapabilities2KHR* pCapabilities);
```

- `physicalDevice` is the physical device associated with `pDisplayPlaneInfo`.
- `pDisplayPlaneInfo` is a pointer to a `VkDisplayPlaneInfo2KHR` structure describing the plane and mode.
- `pCapabilities` is a pointer to a `VkDisplayPlaneCapabilities2KHR` structure in which the capabilities are returned.

`vkGetDisplayPlaneCapabilities2KHR` behaves similarly to `vkGetDisplayPlaneCapabilitiesKHR`, with the ability to specify extended inputs via chained input structures, and to return extended information via chained output structures.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pDisplayPlaneInfo` **must** be a valid pointer to a valid `VkDisplayPlaneInfo2KHR` structure
- `pCapabilities` **must** be a valid pointer to a `VkDisplayPlaneCapabilities2KHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPlaneInfo2KHR` structure is defined as:

```
typedef struct VkDisplayPlaneInfo2KHR {
    VkStructureType    sType;
    const void*        pNext;
    VkDisplayModeKHR   mode;
    uint32_t           planeIndex;
} VkDisplayPlaneInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `mode` is the display mode the application intends to program when using the specified plane.



#### Note

This parameter also implicitly specifies a display.

- `planeIndex` is the plane which the application intends to use with the display.

The members of `VkDisplayPlaneInfo2KHR` correspond to the arguments to `vkGetDisplayPlaneCapabilitiesKHR`, with `sType` and `pNext` added for extensibility.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR`
- `pNext` **must** be `NULL`
- `mode` **must** be a valid `VkDisplayModeKHR` handle

## Host Synchronization

- Host access to `mode` **must** be externally synchronized

The `VkDisplayPlaneCapabilities2KHR` structure is defined as:

```
typedef struct VkDisplayPlaneCapabilities2KHR {
    VkStructureType sType;
    void* pNext;
    VkDisplayPlaneCapabilitiesKHR capabilities;
} VkDisplayPlaneCapabilities2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `capabilities` is an instance of the `VkDisplayPlaneCapabilitiesKHR` structure.

### Valid Usage (Implicit)

- `sType must` be `VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR`
- `pNext must` be `NULL`

## 32.3.2. Display Control

To set the power state of a display, call:

```
VkResult vkDisplayPowerControlEXT(
    VkDevice device,
    VkDisplayKHR display,
    const VkDisplayPowerInfoEXT* pDisplayPowerInfo);
```

- `device` is a logical device associated with `display`.
- `display` is the display whose power state is modified.
- `pDisplayPowerInfo` is an instance of `VkDisplayPowerInfoEXT` specifying the new power state of `display`.

### Valid Usage (Implicit)

- `device must` be a valid `VkDevice` handle
- `display must` be a valid `VkDisplayKHR` handle
- `pDisplayPowerInfo must` be a valid pointer to a valid `VkDisplayPowerInfoEXT` structure

## Return Codes

### Success

- `VK_SUCCESS`

The `VkDisplayPowerInfoEXT` structure is defined as:

```
typedef struct VkDisplayPowerInfoEXT {
    VkStructureType         sType;
    const void*             pNext;
    VkDisplayPowerStateEXT powerState;
} VkDisplayPowerInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `powerState` is a `VkDisplayPowerStateEXT` value specifying the new power state of the display.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT`
- `pNext` **must** be `NULL`
- `powerState` **must** be a valid `VkDisplayPowerStateEXT` value

Possible values of `VkDisplayPowerInfoEXT::powerState`, specifying the new power state of a display, are:

```
typedef enum VkDisplayPowerStateEXT {
    VK_DISPLAY_POWER_STATE_OFF_EXT = 0,
    VK_DISPLAY_POWER_STATE_SUSPEND_EXT = 1,
    VK_DISPLAY_POWER_STATE_ON_EXT = 2,
    VK_DISPLAY_POWER_STATE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDisplayPowerStateEXT;
```

- `VK_DISPLAY_POWER_STATE_OFF_EXT` specifies that the display is powered down.
- `VK_DISPLAY_POWER_STATE_SUSPEND_EXT` specifies that the display is put into a low power mode, from which it **may** be able to transition back to `VK_DISPLAY_POWER_STATE_ON_EXT` more quickly than if it were in `VK_DISPLAY_POWER_STATE_OFF_EXT`. This state **may** be the same as `VK_DISPLAY_POWER_STATE_OFF_EXT`.
- `VK_DISPLAY_POWER_STATE_ON_EXT` specifies that the display is powered on.

### 32.3.3. Display Surfaces

A complete display configuration includes a mode, one or more display planes and any parameters describing their behavior, and parameters describing some aspects of the images associated with those planes. Display surfaces describe the configuration of a single plane within a complete display configuration. To create a `VkSurfaceKHR` structure for a display surface, call:

```
VkResult vkCreateDisplayPlaneSurfaceKHR(  
    VkInstance instance,  
    const VkDisplaySurfaceCreateInfoKHR* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkSurfaceKHR* pSurface);
```

- **instance** is the instance corresponding to the physical device the targeted display is on.
- **pCreateInfo** is a pointer to a `VkDisplaySurfaceCreateInfoKHR` structure specifying which mode, plane, and other parameters to use, as described below.
- **pAllocator** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- **pSurface** is a pointer to a `VkSurfaceKHR` handle in which the created surface is returned.

## Valid Usage (Implicit)

- **instance must** be a valid `VkInstance` handle
- **pCreateInfo must** be a valid pointer to a valid `VkDisplaySurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator must` be a valid pointer to a valid `VkAllocationCallbacks` structure
- **pSurface must** be a valid pointer to a `VkSurfaceKHR` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplaySurfaceCreateInfoKHR` structure is defined as:

```

typedef struct VkDisplaySurfaceCreateInfoKHR {
    VkStructureType           sType;
    const void*             pNext;
    VkDisplaySurfaceCreateFlagsKHR flags;
    VkDisplayModeKHR          displayMode;
    uint32_t                planeIndex;
    uint32_t                planeStackIndex;
    VkSurfaceTransformFlagBitsKHR transform;
    float                  globalAlpha;
    VkDisplayPlaneAlphaFlagBitsKHR alphaMode;
    VkExtent2D                imageExtent;
} VkDisplaySurfaceCreateInfoKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **flags** is reserved for future use, and **must** be zero.
- **displayMode** is a [VkDisplayModeKHR](#) handle specifying the mode to use when displaying this surface.
- **planeIndex** is the plane on which this surface appears.
- **planeStackIndex** is the z-order of the plane.
- **transform** is a [VkSurfaceTransformFlagBitsKHR](#) value specifying the transformation to apply to images as part of the scanout operation.
- **globalAlpha** is the global alpha value. This value is ignored if **alphaMode** is not [VK\\_DISPLAY\\_PLANE\\_ALPHA\\_GLOBAL\\_BIT\\_KHR](#).
- **alphaMode** is a [VkDisplayPlaneAlphaFlagBitsKHR](#) value specifying the type of alpha blending to use.
- **imageExtent** The size of the presentable images to use with the surface.

*Note*



Creating a display surface **must** not modify the state of the displays, planes, or other resources it names. For example, it **must** not apply the specified mode to be set on the associated display. Application of display configuration occurs as a side effect of presenting to a display surface.

## Valid Usage

- `planeIndex` **must** be less than the number of display planes supported by the device as determined by calling `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`
- If the `planeReorderPossible` member of the `VkDisplayPropertiesKHR` structure returned by `vkGetPhysicalDeviceDisplayPropertiesKHR` for the display corresponding to `displayMode` is `VK_TRUE` then `planeStackIndex` **must** be less than the number of display planes supported by the device as determined by calling `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`; otherwise `planeStackIndex` **must** equal the `currentStackIndex` member of `VkDisplayPlanePropertiesKHR` returned by `vkGetPhysicalDeviceDisplayPlanePropertiesKHR` for the display plane corresponding to `displayMode`
- If `alphaMode` is `VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR` then `globalAlpha` **must** be between `0` and `1`, inclusive
- `alphaMode` **must** be `0` or one of the bits present in the `supportedAlpha` member of `VkDisplayPlaneCapabilitiesKHR` returned by `vkGetDisplayPlaneCapabilitiesKHR` for the display plane corresponding to `displayMode`
- The `width` and `height` members of `imageExtent` **must** be less than the `maxImageDimensions2D` member of `VkPhysicalDeviceLimits`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `displayMode` **must** be a valid `VkDisplayModeKHR` handle
- `transform` **must** be a valid `VkSurfaceTransformFlagBitsKHR` value
- `alphaMode` **must** be a valid `VkDisplayPlaneAlphaFlagBitsKHR` value

```
typedef VkFlags VkDisplaySurfaceCreateFlagsKHR;
```

`VkDisplaySurfaceCreateFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.

Possible values of `VkDisplaySurfaceCreateInfoKHR::alphaMode`, specifying the type of alpha blending to use on a display, are:

```
typedef enum VkDisplayPlaneAlphaFlagBitsKHR {
    VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR = 0x00000001,
    VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR = 0x00000002,
    VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR = 0x00000004,
    VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_PREMULTIPLIED_BIT_KHR = 0x00000008,
    VK_DISPLAY_PLANE_ALPHA_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkDisplayPlaneAlphaFlagBitsKHR;
```

- `VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR` specifies that the source image will be treated as opaque.
- `VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR` specifies that a global alpha value **must** be specified that will be applied to all pixels in the source image.
- `VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR` specifies that the alpha value will be determined by the alpha channel of the source image's pixels. If the source format contains no alpha values, no blending will be applied. The source alpha values are not premultiplied into the source image's other color channels.
- `VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_PREMULTIPLIED_BIT_KHR` is equivalent to `VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR`, except the source alpha values are assumed to be premultiplied into the source image's other color channels.

```
typedef VkFlags VkDisplayPlaneAlphaFlagsKHR;
```

`VkDisplayPlaneAlphaFlagsKHR` is a bitmask type for setting a mask of zero or more `VkDisplayPlaneAlphaFlagBitsKHR`.

### 32.3.4. Presenting to headless surfaces

Vulkan rendering can be presented to a headless surface, where the presentation operation is a no-op producing no externally-visible result.

#### Note

Because there is no real presentation target, the headless presentation engine may be extended to impose an arbitrary or customisable set of restrictions and features. This makes it a useful portable test target for applications targeting a wide range of presentation engines where the actual target presentation engines might be scarce, unavailable or otherwise undesirable or inconvenient to use for general Vulkan application development.

The usual surface query mechanisms must be used to determine the actual restrictions and features of the implementation.

To create a headless `VkSurfaceKHR` object, call:



```

VkResult vkCreateHeadlessSurfaceEXT(
    VkInstance instance,
    const VkHeadlessSurfaceCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);

```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkHeadlessSurfaceCreateInfoEXT` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

### Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkHeadlessSurfaceCreateInfoEXT` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

### Return Codes

#### Success

- `VK_SUCCESS`

#### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkHeadlessSurfaceCreateInfoEXT` structure is defined as:

```

typedef struct VkHeadlessSurfaceCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkHeadlessSurfaceCreateFlagsEXT flags;
} VkHeadlessSurfaceCreateInfoEXT;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

For headless surfaces, `currentExtent` is undefined (0xFFFFFFFF, 0xFFFFFFFF). Whatever the application sets a swapchain's `imageExtent` to will be the size of the surface, after the first image is presented.

## 32.4. Querying for WSI Support

Not all physical devices will include WSI support. Within a physical device, not all queue families will support presentation. WSI support and compatibility **can** be determined in a platform-neutral manner (which determines support for presentation to a particular surface object) and additionally **may** be determined in platform-specific manners (which determine support for presentation on the specified physical device but do not guarantee support for presentation to a particular surface object).

To determine whether a queue family of a physical device supports presentation to a given surface, call:

```
VkResult vkGetPhysicalDeviceSurfaceSupportKHR(  
    VkPhysicalDevice                          physicalDevice,  
    uint32_t                                  queueFamilyIndex,  
    VkSurfaceKHR                             surface,  
    VkBool32*                                pSupported);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family.
- `surface` is the surface.
- `pSupported` is a pointer to a `VkBool32`, which is set to `VK_TRUE` to indicate support, and `VK_FALSE` otherwise.

## Valid Usage

- `queueFamilyIndex` **must** be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pSupported` **must** be a valid pointer to a `VkBool32` value
- Both of `physicalDevice`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

### 32.4.1. Android Platform

On Android, all physical devices and queue families **must** be capable of presentation with any native window. As a result there is no Android-specific query for these capabilities.

### 32.4.2. Wayland Platform

To determine whether a queue family of a physical device supports presentation to a Wayland compositor, call:

```
VkBool32 vkGetPhysicalDeviceWaylandPresentationSupportKHR(  
    VkPhysicalDevice                physicalDevice,  
    uint32_t                         queueFamilyIndex,  
    struct wl_display*               display);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.
- `display` is a pointer to the `wl_display` associated with a Wayland compositor.

This platform-specific function **can** be called prior to creating a surface.

## Valid Usage

- `queueFamilyIndex` **must** be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `display` **must** be a valid pointer to a `wl_display` value

### 32.4.3. Win32 Platform

To determine whether a queue family of a physical device supports presentation to the Microsoft Windows desktop, call:

```
VkBool32 vkGetPhysicalDeviceWin32PresentationSupportKHR(  
    VkPhysicalDevice           physicalDevice,  
    uint32_t                   queueFamilyIndex);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.

This platform-specific function **can** be called prior to creating a surface.

## Valid Usage

- `queueFamilyIndex` **must** be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle

### 32.4.4. XCB Platform

To determine whether a queue family of a physical device supports presentation to an X11 server, using the XCB client-side library, call:

```
VkBool32 vkGetPhysicalDeviceXcbPresentationSupportKHR(  
    VkPhysicalDevice physicalDevice,  
    uint32_t queueFamilyIndex,  
    xcb_connection_t* connection,  
    xcb_visualid_t visual_id);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.
- `connection` is a pointer to an `xcb_connection_t` to the X server. `visual_id` is an X11 visual (`xcb_visualid_t`).

This platform-specific function **can** be called prior to creating a surface.

### Valid Usage

- `queueFamilyIndex` **must** be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `connection` **must** be a valid pointer to an `xcb_connection_t` value

## 32.4.5. Xlib Platform

To determine whether a queue family of a physical device supports presentation to an X11 server, using the Xlib client-side library, call:

```
VkBool32 vkGetPhysicalDeviceXlibPresentationSupportKHR(  
    VkPhysicalDevice physicalDevice,  
    uint32_t queueFamilyIndex,  
    Display* dpy,  
    VisualID visualID);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.
- `dpy` is a pointer to an Xlib `Display` connection to the server.
- `visualID` is an X11 visual (`VisualID`).

This platform-specific function **can** be called prior to creating a surface.

## Valid Usage

- `queueFamilyIndex` **must** be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `dpy` **must** be a valid pointer to a `Display` value

### 32.4.6. Fuchsia Platform

On Fuchsia, all physical devices and queue families **must** be capable of presentation with any ImagePipe. As a result there is no Fuchsia-specific query for these capabilities.

### 32.4.7. Google Games Platform

On Google Games Platform, all physical devices and queue families with the `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT` capabilities **must** be capable of presentation with any Google Games Platform stream descriptor. As a result, there is no query specific to Google Games Platform for these capabilities.

### 32.4.8. iOS Platform

On iOS, all physical devices and queue families **must** be capable of presentation with any layer. As a result there is no iOS-specific query for these capabilities.

### 32.4.9. macOS Platform

On macOS, all physical devices and queue families **must** be capable of presentation with any layer. As a result there is no macOS-specific query for these capabilities.

### 32.4.10. VI Platform

On VI, all physical devices and queue families **must** be capable of presentation with any layer. As a result there is no VI-specific query for these capabilities.

## 32.5. Surface Queries

The capabilities of a swapchain targeting a surface are the intersection of the capabilities of the WSI platform, the native window or display, and the physical device. The resulting capabilities can be obtained with the queries listed below in this section. Capabilities that correspond to image creation parameters are not independent of each other: combinations of parameters that are not supported as reported by `vkGetPhysicalDeviceImageFormatProperties` are not supported by the surface on that physical device, even if the capabilities taken individually are supported as part of

some other parameter combinations.

### 32.5.1. Surface Capabilities

To query the basic capabilities of a surface, needed in order to create a swapchain, call:

```
VkResult vkGetPhysicalDeviceSurfaceCapabilitiesKHR(  
    VkPhysicalDevice physicalDevice,  
    VkSurfaceKHR surface,  
    VkSurfaceCapabilitiesKHR* pSurfaceCapabilities);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for [vkCreateSwapchainKHR](#).
- `surface` is the surface that will be associated with the swapchain.
- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilitiesKHR` structure in which the capabilities are returned.

#### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pSurfaceCapabilities` **must** be a valid pointer to a `VkSurfaceCapabilitiesKHR` structure
- Both of `physicalDevice`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

#### Return Codes

##### Success

- `VK_SUCCESS`

##### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkSurfaceCapabilitiesKHR` structure is defined as:

```

typedef struct VkSurfaceCapabilitiesKHR {
    uint32_t minImageCount;
    uint32_t maxImageCount;
    VkExtent2D currentExtent;
    VkExtent2D minImageExtent;
    VkExtent2D maxImageExtent;
    uint32_t maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkSurfaceTransformFlagBitsKHR currentTransform;
    VkCompositeAlphaFlagsKHR supportedCompositeAlpha;
    VkImageUsageFlags supportedUsageFlags;
} VkSurfaceCapabilitiesKHR;

```

- **minImageCount** is the minimum number of images the specified device supports for a swapchain created for the surface, and will be at least one.
- **maxImageCount** is the maximum number of images the specified device supports for a swapchain created for the surface, and will be either 0, or greater than or equal to **minImageCount**. A value of 0 means that there is no limit on the number of images, though there **may** be limits related to the total amount of memory used by presentable images.
- **currentExtent** is the current width and height of the surface, or the special value (0xFFFFFFFF, 0xFFFFFFFF) indicating that the surface size will be determined by the extent of a swapchain targeting the surface.
- **minImageExtent** contains the smallest valid swapchain extent for the surface on the specified device. The **width** and **height** of the extent will each be less than or equal to the corresponding **width** and **height** of **currentExtent**, unless **currentExtent** has the special value described above.
- **maxImageExtent** contains the largest valid swapchain extent for the surface on the specified device. The **width** and **height** of the extent will each be greater than or equal to the corresponding **width** and **height** of **minImageExtent**. The **width** and **height** of the extent will each be greater than or equal to the corresponding **width** and **height** of **currentExtent**, unless **currentExtent** has the special value described above.
- **maxImageArrayLayers** is the maximum number of layers presentable images **can** have for a swapchain created for this device and surface, and will be at least one.
- **supportedTransforms** is a bitmask of **VkSurfaceTransformFlagBitsKHR** indicating the presentation transforms supported for the surface on the specified device. At least one bit will be set.
- **currentTransform** is **VkSurfaceTransformFlagBitsKHR** value indicating the surface's current transform relative to the presentation engine's natural orientation.
- **supportedCompositeAlpha** is a bitmask of **VkCompositeAlphaFlagBitsKHR**, representing the alpha compositing modes supported by the presentation engine for the surface on the specified device, and at least one bit will be set. Opaque composition **can** be achieved in any alpha compositing mode by either using an image format that has no alpha component, or by ensuring that all pixels in the presentable images have an alpha value of 1.0.
- **supportedUsageFlags** is a bitmask of **VkImageUsageFlagBits** representing the ways the application **can** use the presentable images of a swapchain created with **VkPresentModeKHR** set

to `VK_PRESENT_MODE_IMMEDIATE_KHR`, `VK_PRESENT_MODE_MAILBOX_KHR`, `VK_PRESENT_MODE_FIFO_KHR` or `VK_PRESENT_MODE_FIFO_RELAXED_KHR` for the surface on the specified device. `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` must be included in the set but implementations may support additional usages.

*Note*



Supported usage flags of a presentable image when using `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` presentation mode are provided by `VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags`.

*Note*



Formulas such as  $\min(N, \text{maxImageCount})$  are not correct, since `maxImageCount` may be zero.

To query the basic capabilities of a surface defined by the core or extensions, call:

```
VkResult vkGetPhysicalDeviceSurfaceCapabilities2KHR(  
    VkPhysicalDevice physicalDevice,  
    const VkPhysicalDeviceSurfaceInfo2KHR* pSurfaceInfo,  
    VkSurfaceCapabilities2KHR* pSurfaceCapabilities);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `pSurfaceInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by `vkCreateSwapchainKHR`.
- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilities2KHR` structure in which the capabilities are returned.

`vkGetPhysicalDeviceSurfaceCapabilities2KHR` behaves similarly to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with the ability to specify extended inputs via chained input structures, and to return extended information via chained output structures.

## Valid Usage

- If an instance of `VkSurfaceCapabilitiesFullScreenExclusiveEXT` is included in the `pNext` chain of `pSurfaceCapabilities`, an instance of `VkSurfaceFullScreenExclusiveWin32InfoEXT` must be included in the `pNext` chain of `pSurfaceInfo`.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pSurfaceInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceSurfaceInfo2KHR` structure
- `pSurfaceCapabilities` **must** be a valid pointer to a `VkSurfaceCapabilities2KHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkPhysicalDeviceSurfaceInfo2KHR` structure is defined as:

```
typedef struct VkPhysicalDeviceSurfaceInfo2KHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSurfaceKHR       surface;
} VkPhysicalDeviceSurfaceInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `surface` is the surface that will be associated with the swapchain.

The members of `VkPhysicalDeviceSurfaceInfo2KHR` correspond to the arguments to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with `sType` and `pNext` added for extensibility.

Additional capabilities of a surface **may** be available to swapchains created with different full-screen exclusive settings - particularly if exclusive full-screen access is application controlled. These additional capabilities **can** be queried by including the `VkSurfaceFullScreenExclusiveInfoEXT` structure in the `pNext` chain of this structure when used to query surface properties. Additionally, for Win32 surfaces with application controlled exclusive full-screen access, chaining a valid instance of the `VkSurfaceFullScreenExclusiveWin32InfoEXT` structure **may** also report additional surface capabilities. These additional capabilities only apply to swapchains created with the same parameters passed into the `pNext` chain of `VkSwapchainCreateInfoKHR`.

## Valid Usage

- If the `pNext` chain includes an instance of `VkSurfaceFullScreenExclusiveInfoEXT` with its `fullScreenExclusive` member set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`, and `surface` was created using `vkCreateWin32SurfaceKHR`, an instance of `VkSurfaceFullScreenExclusiveWin32InfoEXT` must be present in the `pNext` chain

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkSurfaceFullScreenExclusiveInfoEXT` or `VkSurfaceFullScreenExclusiveWin32InfoEXT`
- Each `sType` member in the `pNext` chain must be unique
- `surface` must be a valid `VkSurfaceKHR` handle

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkSurfaceFullScreenExclusiveInfoEXT` structure, then that structure specifies the application's preferred full-screen transition behavior.

The `VkSurfaceFullScreenExclusiveInfoEXT` structure is defined as:

```
typedef struct VkSurfaceFullScreenExclusiveInfoEXT {
    VkStructureType           sType;
    void*                     pNext;
    VkFullScreenExclusiveEXT  fullScreenExclusive;
} VkSurfaceFullScreenExclusiveInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `fullScreenExclusive` is a `VkFullScreenExclusiveEXT` value specifying the preferred full-screen transition behavior.

If this structure is not present, `fullScreenExclusive` is considered to be `VK_FULL_SCREEN_EXCLUSIVE_DEFAULT_EXT`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SURFACE_FULLSCREEN_EXCLUSIVE_INFO_EXT`
- `fullScreenExclusive` must be a valid `VkFullScreenExclusiveEXT` value

Possible values of `VkSurfaceFullScreenExclusiveInfoEXT::fullScreenExclusive` are:

```

typedef enum VkFullScreenExclusiveEXT {
    VK_FULLSCREEN_EXCLUSIVE_DEFAULT_EXT = 0,
    VK_FULLSCREEN_EXCLUSIVE_ALLOWED_EXT = 1,
    VK_FULLSCREEN_EXCLUSIVE_DISALLOWED_EXT = 2,
    VK_FULLSCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT = 3,
    VK_FULLSCREEN_EXCLUSIVE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkFullScreenExclusiveEXT;

```

- `VK_FULLSCREEN_EXCLUSIVE_DEFAULT_EXT` indicates the implementation **should** determine the appropriate full-screen method by whatever means it deems appropriate.
- `VK_FULLSCREEN_EXCLUSIVE_ALLOWED_EXT` indicates the implementation **may** use full-screen exclusive mechanisms when available. Such mechanisms **may** result in better performance and/or the availability of different presentation capabilities, but **may** require a more disruptive transition during swapchain initialization, first presentation and/or destruction.
- `VK_FULLSCREEN_EXCLUSIVE_DISALLOWED_EXT` indicates the implementation **should** avoid using full-screen mechanisms which rely on disruptive transitions.
- `VK_FULLSCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT` indicates the application will manage full-screen exclusive mode by using the `vkAcquireFullScreenExclusiveModeEXT` and `vkReleaseFullScreenExclusiveModeEXT` commands.

The `VkSurfaceFullScreenExclusiveWin32InfoEXT` structure is defined as:

```

typedef struct VkSurfaceFullScreenExclusiveWin32InfoEXT {
    VkStructureType      sType;
    const void*        pNext;
    HMONITOR            hmonitor;
} VkSurfaceFullScreenExclusiveWin32InfoEXT;

```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `hmonitor` is the Win32 `HMONITOR` handle identifying the display to create the surface with.

*Note*



If `hmonitor` is invalidated (e.g. the monitor is unplugged) during the lifetime of a swapchain created with this structure, operations on that swapchain will return `VK_ERROR_OUT_OF_DATE_KHR`.

*Note*



It's the responsibility of the application to change the display settings of the targeted Win32 display using the appropriate platform APIs. Such changes **may** alter the surface capabilities reported for the created surface.

## Valid Usage

- `hmonitor` **must** be a valid `HMONITOR`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SURFACE_FULL_SCREEN_EXCLUSIVE_WIN32_INFO_EXT`

The `VkSurfaceCapabilities2KHR` structure is defined as:

```
typedef struct VkSurfaceCapabilities2KHR {
    VkStructureType          sType;
    void*                   pNext;
    VkSurfaceCapabilitiesKHR surfaceCapabilities;
} VkSurfaceCapabilities2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `surfaceCapabilities` is a `VkSurfaceCapabilitiesKHR` structure describing the capabilities of the specified surface.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkDisplayNativeHdrSurfaceCapabilitiesAMD`, `VkSharedPresentSurfaceCapabilitiesKHR`, `VkSurfaceCapabilitiesFullScreenExclusiveEXT`, or `VkSurfaceProtectedCapabilitiesKHR`
- Each `sType` member in the `pNext` chain **must** be unique

An application queries if a protected `VkSurfaceKHR` is displayable on a specific windowing system using `VkSurfaceProtectedCapabilitiesKHR`, which `can` be passed in `pNext` parameter of `VkSurfaceCapabilities2KHR`.

The `VkSurfaceProtectedCapabilitiesKHR` structure is defined as:

```
typedef struct VkSurfaceProtectedCapabilitiesKHR {
    VkStructureType      sType;
    const void*          pNext;
    VkBool32              supportsProtected;
} VkSurfaceProtectedCapabilitiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `supportsProtected` specifies whether a protected swapchain created from `VkPhysicalDeviceSurfaceInfo2KHR::surface` for a particular windowing system **can** be displayed on screen or not. If `supportsProtected` is `VK_TRUE`, then creation of swapchains with the `VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR` flag set **must** be supported for `surface`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SURFACE_PROTECTED_CAPABILITIES_KHR`

The `VkSharedPresentSurfaceCapabilitiesKHR` structure is defined as:

```
typedef struct VkSharedPresentSurfaceCapabilitiesKHR {
    VkStructureType      sType;
    void*                pNext;
    VkImageUsageFlags    sharedPresentSupportedUsageFlags;
} VkSharedPresentSurfaceCapabilitiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `sharedPresentSupportedUsageFlags` is a bitmask of `VkImageUsageFlagBits` representing the ways the application **can** use the shared presentable image from a swapchain created with `VkPresentModeKHR` set to `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` for the surface on the specified device. `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` **must** be included in the set but implementations **may** support additional usages.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR`

The `VkDisplayNativeHdrSurfaceCapabilitiesAMD` structure is defined as:

```
typedef struct VkDisplayNativeHdrSurfaceCapabilitiesAMD {
    VkStructureType      sType;
    void*                pNext;
    VkBool32              localDimmingSupport;
} VkDisplayNativeHdrSurfaceCapabilitiesAMD;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `localDimmingSupport` specifies whether the surface supports local dimming. If this is `VK_TRUE`, `VkSwapchainDisplayNativeHdrCreateInfoAMD` can be used to explicitly enable or disable local dimming for the surface. Local dimming may also be overridden by `vkSetLocalDimmingAMD` during the lifetime of the swapchain.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DISPLAY_NATIVE_HDR_SURFACE_CAPABILITIES_AMD`

The `VkSurfaceCapabilitiesFullScreenExclusiveEXT` structure is defined as:

```
typedef struct VkSurfaceCapabilitiesFullScreenExclusiveEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            fullScreenExclusiveSupported;
} VkSurfaceCapabilitiesFullScreenExclusiveEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `fullScreenExclusiveControlSupported` is a boolean describing whether the surface is able to make use of exclusive full-screen access.

This structure can be included in the `pNext` chain of `VkSurfaceCapabilities2KHR` to determine support for exclusive full-screen access. If `fullScreenExclusiveSupported` is `VK_FALSE`, it indicates that exclusive full-screen access is not obtainable for this surface.

Applications must not attempt to create swapchains with `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT` set if `fullScreenExclusiveSupported` is `VK_FALSE`.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_FULLSCREEN_EXCLUSIVE_EXT`

To query the basic capabilities of a surface, needed in order to create a swapchain, call:

```
VkResult vkGetPhysicalDeviceSurfaceCapabilities2EXT(
    VkPhysicalDevice                          physicalDevice,
    VkSurfaceKHR                            surface,
    VkSurfaceCapabilities2EXT*               pSurfaceCapabilities);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `surface` is the surface that will be associated with the swapchain.

- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilities2EXT` structure in which the capabilities are returned.

`vkGetPhysicalDeviceSurfaceCapabilities2EXT` behaves similarly to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with the ability to return extended information by adding extension structures to the `pNext` chain of its `pSurfaceCapabilities` parameter.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pSurfaceCapabilities` **must** be a valid pointer to a `VkSurfaceCapabilities2EXT` structure
- Both of `physicalDevice`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkSurfaceCapabilities2EXT` structure is defined as:

```
typedef struct VkSurfaceCapabilities2EXT {
    VkStructureType           sType;
    void*                     pNext;
    uint32_t                  minImageCount;
    uint32_t                  maxImageCount;
    VkExtent2D                 currentExtent;
    VkExtent2D                 minImageExtent;
    VkExtent2D                 maxImageExtent;
    uint32_t                  maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkSurfaceTransformFlagBitsKHR currentTransform;
    VkCompositeAlphaFlagsKHR   supportedCompositeAlpha;
    VkImageUsageFlags          supportedUsageFlags;
    VkSurfaceCounterFlagsEXT   supportedSurfaceCounters;
} VkSurfaceCapabilities2EXT;
```

All members of `VkSurfaceCapabilities2EXT` are identical to the corresponding members of `VkSurfaceCapabilitiesKHR` where one exists. The remaining members are:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `supportedSurfaceCounters` is a bitmask of `VkSurfaceCounterFlagBitsEXT` indicating the supported surface counter types.

## Valid Usage

- `supportedSurfaceCounters` **must** not include `VK_SURFACE_COUNTER_VBLANK_EXT` unless the surface queried is a [display surface](#).

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT`
- `pNext` **must** be `NULL`

Bits which **can** be set in `VkSurfaceCapabilities2EXT::supportedSurfaceCounters`, indicating supported surface counter types, are:

```
typedef enum VkSurfaceCounterFlagBitsEXT {
    VK_SURFACE_COUNTER_VBLANK_EXT = 0x00000001,
    VK_SURFACE_COUNTER_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkSurfaceCounterFlagBitsEXT;
```

- `VK_SURFACE_COUNTER_VBLANK_EXT` specifies a counter incrementing once every time a vertical blanking period occurs on the display associated with the surface.

```
typedef VkFlags VkSurfaceCounterFlagsEXT;
```

`VkSurfaceCounterFlagsEXT` is a bitmask type for setting a mask of zero or more `VkSurfaceCounterFlagBitsEXT`.

Bits which **may** be set in `VkSurfaceCapabilitiesKHR::supportedTransforms` indicating the presentation transforms supported for the surface on the specified device, and possible values of `VkSurfaceCapabilitiesKHR::currentTransform` is indicating the surface's current transform relative to the presentation engine's natural orientation, are:

```

typedef enum VkSurfaceTransformFlagBitsKHR {
    VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR = 0x00000001,
    VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR = 0x00000002,
    VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR = 0x00000004,
    VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR = 0x00000008,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR = 0x00000010,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR = 0x00000020,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR = 0x00000040,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR = 0x00000080,
    VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR = 0x00000100,
    VK_SURFACE_TRANSFORM_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkSurfaceTransformFlagBitsKHR;

```

- **VK\_SURFACE\_TRANSFORM\_IDENTITY\_BIT\_KHR** specifies that image content is presented without being transformed.
- **VK\_SURFACE\_TRANSFORM\_ROTATE\_90\_BIT\_KHR** specifies that image content is rotated 90 degrees clockwise.
- **VK\_SURFACE\_TRANSFORM\_ROTATE\_180\_BIT\_KHR** specifies that image content is rotated 180 degrees clockwise.
- **VK\_SURFACE\_TRANSFORM\_ROTATE\_270\_BIT\_KHR** specifies that image content is rotated 270 degrees clockwise.
- **VK\_SURFACE\_TRANSFORM\_HORIZONTAL\_MIRROR\_BIT\_KHR** specifies that image content is mirrored horizontally.
- **VK\_SURFACE\_TRANSFORM\_HORIZONTAL\_MIRROR\_ROTATE\_90\_BIT\_KHR** specifies that image content is mirrored horizontally, then rotated 90 degrees clockwise.
- **VK\_SURFACE\_TRANSFORM\_HORIZONTAL\_MIRROR\_ROTATE\_180\_BIT\_KHR** specifies that image content is mirrored horizontally, then rotated 180 degrees clockwise.
- **VK\_SURFACE\_TRANSFORM\_HORIZONTAL\_MIRROR\_ROTATE\_270\_BIT\_KHR** specifies that image content is mirrored horizontally, then rotated 270 degrees clockwise.
- **VK\_SURFACE\_TRANSFORM\_INHERIT\_BIT\_KHR** specifies that the presentation transform is not specified, and is instead determined by platform-specific considerations and mechanisms outside Vulkan.

```
typedef VkFlags VkSurfaceTransformFlagsKHR;
```

`VkSurfaceTransformFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSurfaceTransformFlagBitsKHR`.

The `supportedCompositeAlpha` member is of type `VkCompositeAlphaFlagBitsKHR`, which contains the following values:

```

typedef enum VkCompositeAlphaFlagBitsKHR {
    VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR = 0x00000001,
    VK_COMPOSITE_ALPHA_PRE_MULTIPLIED_BIT_KHR = 0x00000002,
    VK_COMPOSITE_ALPHA_POST_MULTIPLIED_BIT_KHR = 0x00000004,
    VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR = 0x00000008,
    VK_COMPOSITE_ALPHA_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkCompositeAlphaFlagBitsKHR;

```

These values are described as follows:

- **VK\_COMPOSITE\_ALPHA\_OPAQUE\_BIT\_KHR**: The alpha channel, if it exists, of the images is ignored in the compositing process. Instead, the image is treated as if it has a constant alpha of 1.0.
- **VK\_COMPOSITE\_ALPHA\_PRE\_MULTIPLIED\_BIT\_KHR**: The alpha channel, if it exists, of the images is respected in the compositing process. The non-alpha channels of the image are expected to already be multiplied by the alpha channel by the application.
- **VK\_COMPOSITE\_ALPHA\_POST\_MULTIPLIED\_BIT\_KHR**: The alpha channel, if it exists, of the images is respected in the compositing process. The non-alpha channels of the image are not expected to already be multiplied by the alpha channel by the application; instead, the compositor will multiply the non-alpha channels of the image by the alpha channel during compositing.
- **VK\_COMPOSITE\_ALPHA\_INHERIT\_BIT\_KHR**: The way in which the presentation engine treats the alpha channel in the images is unknown to the Vulkan API. Instead, the application is responsible for setting the composite alpha blending mode using native window system commands. If the application does not set the blending mode using native window system commands, then a platform-specific default will be used.

```
typedef VkFlags VkCompositeAlphaFlagsKHR;
```

`VkCompositeAlphaFlagsKHR` is a bitmask type for setting a mask of zero or more `VkCompositeAlphaFlagBitsKHR`.

### 32.5.2. Surface Format Support

To query the supported swapchain format-color space pairs for a surface, call:

```

VkResult vkGetPhysicalDeviceSurfaceFormatsKHR(
    VkPhysicalDevice physicalDevice,
    VkSurfaceKHR surface,
    uint32_t* pSurfaceFormatCount,
    VkSurfaceFormatKHR** pSurfaceFormats);

```

- **physicalDevice** is the physical device that will be associated with the swapchain to be created, as described for [vkCreateSwapchainKHR](#).
- **surface** is the surface that will be associated with the swapchain.
- **pSurfaceFormatCount** is a pointer to an integer related to the number of format pairs available or

queried, as described below.

- `pSurfaceFormats` is either `NULL` or a pointer to an array of `VkSurfaceFormatKHR` structures.

If `pSurfaceFormats` is `NULL`, then the number of format pairs supported for the given `surface` is returned in `pSurfaceFormatCount`. Otherwise, `pSurfaceFormatCount` **must** point to a variable set by the user to the number of elements in the `pSurfaceFormats` array, and on return the variable is overwritten with the number of structures actually written to `pSurfaceFormats`. If the value of `pSurfaceFormatCount` is less than the number of format pairs supported, at most `pSurfaceFormatCount` structures will be written. If `pSurfaceFormatCount` is smaller than the number of format pairs supported for the given `surface`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

The number of format pairs supported **must** be greater than or equal to 1. `pSurfaceFormats` **must** not contain an entry whose value for `format` is `VK_FORMAT_UNDEFINED`.

If `pSurfaceFormats` includes an entry whose value for `colorSpace` is `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR` and whose value for `format` is a UNORM (or SRGB) format and the corresponding SRGB (or UNORM) format is a color renderable format for `VK_IMAGE_TILING_OPTIMAL`, then `pSurfaceFormats` **must** also contain an entry with the same value for `colorSpace` and `format` equal to the corresponding SRGB (or UNORM) format.

## Valid Usage

- `surface` **must** be supported by `physicalDevice`, as reported by `vkGetPhysicalDeviceSurfaceSupportKHR` or an equivalent platform-specific mechanism.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pSurfaceFormatCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pSurfaceFormatCount` is not `0`, and `pSurfaceFormats` is not `NULL`, `pSurfaceFormats` **must** be a valid pointer to an array of `pSurfaceFormatCount` `VkSurfaceFormatKHR` structures
- Both of `physicalDevice`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkSurfaceFormatKHR` structure is defined as:

```
typedef struct VkSurfaceFormatKHR {
    VkFormat      format;
    VkColorSpaceKHR colorSpace;
} VkSurfaceFormatKHR;
```

- `format` is a `VkFormat` that is compatible with the specified surface.
- `colorSpace` is a presentation `VkColorSpaceKHR` that is compatible with the surface.

To query the supported swapchain format tuples for a surface, call:

```
VkResult vkGetPhysicalDeviceSurfaceFormats2KHR(
    VkPhysicalDevice           physicalDevice,
    const VkPhysicalDeviceSurfaceInfo2KHR* pCreateInfo,
    uint32_t*                  pSurfaceFormatCount,
    VkSurfaceFormat2KHR*        pSurfaceFormats);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for [vkCreateSwapchainKHR](#).
- `pCreateInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by [vkCreateSwapchainKHR](#).
- `pSurfaceFormatCount` is a pointer to an integer related to the number of format tuples available or queried, as described below.
- `pSurfaceFormats` is either `NULL` or a pointer to an array of `VkSurfaceFormat2KHR` structures.

`vkGetPhysicalDeviceSurfaceFormats2KHR` behaves similarly to `vkGetPhysicalDeviceSurfaceFormatsKHR`, with the ability to be extended via `pNext` chains.

If `pSurfaceFormats` is `NULL`, then the number of format tuples supported for the given `surface` is returned in `pSurfaceFormatCount`. Otherwise, `pSurfaceFormatCount` **must** point to a variable set by the user to the number of elements in the `pSurfaceFormats` array, and on return the variable is overwritten with the number of structures actually written to `pSurfaceFormats`. If the value of

`pSurfaceFormatCount` is less than the number of format tuples supported, at most `pSurfaceFormatCount` structures will be written. If `pSurfaceFormatCount` is smaller than the number of format tuples supported for the surface parameters described in `pSurfaceInfo`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage

- `pSurfaceInfo::surface` must be supported by `physicalDevice`, as reported by `vkGetPhysicalDeviceSurfaceSupportKHR` or an equivalent platform-specific mechanism.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pSurfaceInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceSurfaceInfo2KHR` structure
- `pSurfaceFormatCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pSurfaceFormatCount` is not `0`, and `pSurfaceFormats` is not `NULL`, `pSurfaceFormats` **must** be a valid pointer to an array of `pSurfaceFormatCount` `VkSurfaceFormat2KHR` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkSurfaceFormat2KHR` structure is defined as:

```
typedef struct VkSurfaceFormat2KHR {
    VkStructureType sType;
    void* pNext;
    VkSurfaceFormatKHR surfaceFormat;
} VkSurfaceFormat2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `surfaceFormat` is an instance of `VkSurfaceFormatKHR` describing a format-color space pair that is compatible with the specified surface.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR`
- `pNext` must be `NULL`

While the `format` of a presentable image refers to the encoding of each pixel, the `colorSpace` determines how the presentation engine interprets the pixel values. A color space in this document refers to a specific color space (defined by the chromaticities of its primaries and a white point in CIE Lab), and a transfer function that is applied before storing or transmitting color data in the given color space.

Possible values of `VkSurfaceFormatKHR::colorSpace`, specifying supported color spaces of a presentation engine, are:

```
typedef enum VkColorSpaceKHR {
    VK_COLOR_SPACE_SRGB_NONLINEAR_KHR = 0,
    VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT = 1000104001,
    VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT = 1000104002,
    VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT = 1000104003,
    VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT = 1000104004,
    VK_COLOR_SPACE_BT709_LINEAR_EXT = 1000104005,
    VK_COLOR_SPACE_BT709_NONLINEAR_EXT = 1000104006,
    VK_COLOR_SPACE_BT2020_LINEAR_EXT = 1000104007,
    VK_COLOR_SPACE_HDR10_ST2084_EXT = 1000104008,
    VK_COLOR_SPACE_DOLBYVISION_EXT = 1000104009,
    VK_COLOR_SPACE_HDR10_HLG_EXT = 1000104010,
    VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT = 1000104011,
    VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT = 1000104012,
    VK_COLOR_SPACE_PASS_THROUGH_EXT = 1000104013,
    VK_COLOR_SPACE_EXTENDED_SRGB_NONLINEAR_EXT = 1000104014,
    VK_COLOR_SPACE_DISPLAY_NATIVE_AMD = 1000213000,
    VK_COLORSPACE_SRGB_NONLINEAR_KHR = VK_COLOR_SPACE_SRGB_NONLINEAR_KHR,
    VK_COLOR_SPACE_DCI_P3_LINEAR_EXT = VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT,
    VK_COLOR_SPACE_MAX_ENUM_KHR = 0x7FFFFFFF
} VkColorSpaceKHR;
```

- `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR` specifies support for the sRGB color space.
- `VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT` specifies support for the Display-P3 color space to be displayed using an sRGB-like EOTF (defined below).
- `VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT` specifies support for the extended sRGB color space to be displayed using a linear EOTF.
- `VK_COLOR_SPACE_EXTENDED_SRGB_NONLINEAR_EXT` specifies support for the extended sRGB color space to be displayed using an sRGB EOTF.
- `VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT` specifies support for the Display-P3 color space to be displayed using a linear EOTF.

- `VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT` specifies support for the DCI-P3 color space to be displayed using the DCI-P3 EOTF. Note that values in such an image are interpreted as XYZ encoded color data by the presentation engine.
- `VK_COLOR_SPACE_BT709_LINEAR_EXT` specifies support for the BT709 color space to be displayed using a linear EOTF.
- `VK_COLOR_SPACE_BT709_NONLINEAR_EXT` specifies support for the BT709 color space to be displayed using the SMPTE 170M EOTF.
- `VK_COLOR_SPACE_BT2020_LINEAR_EXT` specifies support for the BT2020 color space to be displayed using a linear EOTF.
- `VK_COLOR_SPACE_HDR10_ST2084_EXT` specifies support for the HDR10 (BT2020 color) space to be displayed using the SMPTE ST2084 Perceptual Quantizer (PQ) EOTF.
- `VK_COLOR_SPACE_DOLBYVISION_EXT` specifies support for the Dolby Vision (BT2020 color space), proprietary encoding, to be displayed using the SMPTE ST2084 EOTF.
- `VK_COLOR_SPACE_HDR10_HLG_EXT` specifies support for the HDR10 (BT2020 color space) to be displayed using the Hybrid Log Gamma (HLG) EOTF.
- `VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT` specifies support for the AdobeRGB color space to be displayed using a linear EOTF.
- `VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT` specifies support for the AdobeRGB color space to be displayed using the Gamma 2.2 EOTF.
- `VK_COLOR_SPACE_PASS_THROUGH_EXT` specifies that color components are used “as is”. This is intended to allow applications to supply data for color spaces not described here.
- `VK_COLOR_SPACE_DISPLAY_NATIVE_AMD` specifies support for the display’s native color space. This matches the color space expectations of AMD’s FreeSync2 standard, for displays supporting it.

*Note*

In the initial release of the `VK_KHR_surface` and `VK_KHR_swapchain` extensions, the token `VK_COLORSPACE_SRGB_NONLINEAR_KHR` was used. Starting in the 2016-05-13 updates to the extension branches, matching release 1.0.13 of the core API specification, `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR` is used instead for consistency with Vulkan naming rules. The older enum is still available for backwards compatibility.

*Note*

In older versions of this extension `VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT` was misnamed `VK_COLOR_SPACE_DCI_P3_LINEAR_EXT`. This has been updated to indicate that it uses RGB color encoding, not XYZ. The old name is deprecated but is maintained for backwards compatibility.

The color components of non-linear color space swap chain images **must** have had the appropriate transfer function applied. The color space selected for the swap chain image will not affect the processing of data written into the image by the implementation. Vulkan requires that all implementations support the sRGB transfer function by use of an SRGB pixel format. Other transfer functions, such as SMPTE 170M or SMPTE2084, **can** be performed by the application shader. This

extension defines enums for [VkColorSpaceKHR](#) that correspond to the following color spaces:

Table 43. Color Spaces and Attributes

Name	Red Primary	Green Primary	Blue Primary	White-point	Transfer function
DCI-P3	1.000, 0.000	0.000, 1.000	0.000, 0.000	0.3333, 0.3333	DCI P3
Display-P3	0.680, 0.320	0.265, 0.690	0.150, 0.060	0.3127, 0.3290 (D65)	Display-P3
BT709	0.640, 0.330	0.300, 0.600	0.150, 0.060	0.3127, 0.3290 (D65)	ITU (SMPTE 170M)
sRGB	0.640, 0.330	0.300, 0.600	0.150, 0.060	0.3127, 0.3290 (D65)	sRGB
extended sRGB	0.640, 0.330	0.300, 0.600	0.150, 0.060	0.3127, 0.3290 (D65)	extended sRGB
HDR10_ST2084	0.708, 0.292	0.170, 0.797	0.131, 0.046	0.3127, 0.3290 (D65)	ST2084 PQ
DOLBYVISION	0.708, 0.292	0.170, 0.797	0.131, 0.046	0.3127, 0.3290 (D65)	ST2084 PQ
HDR10_HLG	0.708, 0.292	0.170, 0.797	0.131, 0.046	0.3127, 0.3290 (D65)	HLG
AdobeRGB	0.640, 0.330	0.210, 0.710	0.150, 0.060	0.3127, 0.3290 (D65)	AdobeRGB

The transfer functions are described in the “Transfer Functions” chapter of the [Khronos Data Format Specification](#).

Except Display-P3 OETF, which is:

$$E = \begin{cases} 1.055 \times L^{\frac{1}{2.4}} - 0.055 & \text{for } 0.0030186 \leq L \leq 1 \\ 12.92 \times L & \text{for } 0 \leq L < 0.0030186 \end{cases}$$

where L is the linear value of a color channel and E is the encoded value (as stored in the image in memory).



*Note*

For most uses, the sRGB OETF is equivalent.

### 32.5.3. Surface Presentation Mode Support

To query the supported presentation modes for a surface, call:

```
VkResult vkGetPhysicalDeviceSurfacePresentModesKHR(  
    VkPhysicalDevice physicalDevice,  
    VkSurfaceKHR surface,  
    uint32_t* pPresentModeCount,  
    VkPresentModeKHR* pPresentModes);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for [vkCreateSwapchainKHR](#).
- `surface` is the surface that will be associated with the swapchain.
- `pPresentModeCount` is a pointer to an integer related to the number of presentation modes available or queried, as described below.
- `pPresentModes` is either `NULL` or a pointer to an array of `VkPresentModeKHR` values, indicating the supported presentation modes.

If `pPresentModes` is `NULL`, then the number of presentation modes supported for the given `surface` is returned in `pPresentModeCount`. Otherwise, `pPresentModeCount` **must** point to a variable set by the user to the number of elements in the `pPresentModes` array, and on return the variable is overwritten with the number of values actually written to `pPresentModes`. If the value of `pPresentModeCount` is less than the number of presentation modes supported, at most `pPresentModeCount` values will be written. If `pPresentModeCount` is smaller than the number of presentation modes supported for the given `surface`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pPresentModeCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPresentModeCount` is not `0`, and `pPresentModes` is not `NULL`, `pPresentModes` **must** be a valid pointer to an array of `pPresentModeCount` `VkPresentModeKHR` values
- Both of `physicalDevice`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

Alternatively, to query the supported presentation modes for a surface combined with select other fixed swapchain creation parameters, call:

```
VkResult vkGetPhysicalDeviceSurfacePresentModes2EXT(  
    VkPhysicalDevice                      physicalDevice,  
    const VkPhysicalDeviceSurfaceInfo2KHR* pSurfaceInfo,  
    uint32_t*                            pPresentModeCount,  
    VkPresentModeKHR*                   pPresentModes);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for [vkCreateSwapchainKHR](#).
- `pSurfaceInfo` is a pointer to a [VkPhysicalDeviceSurfaceInfo2KHR](#) structure describing the surface and other fixed parameters that would be consumed by [vkCreateSwapchainKHR](#).
- `pPresentModeCount` is a pointer to an integer related to the number of presentation modes available or queried, as described below.
- `pPresentModes` is either `NULL` or a pointer to an array of [VkPresentModeKHR](#) values, indicating the supported presentation modes.

`vkGetPhysicalDeviceSurfacePresentModes2EXT` behaves similarly to `vkGetPhysicalDeviceSurfacePresentModesKHR`, with the ability to specify extended inputs via chained input structures.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pSurfaceInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceSurfaceInfo2KHR` structure
- `pPresentModeCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPresentModeCount` is not `0`, and `pPresentModes` is not `NULL`, `pPresentModes` **must** be a valid pointer to an array of `pPresentModeCount` [VkPresentModeKHR](#) values

## Return Codes

### Success

- VK\_SUCCESS
- VK\_INCOMPLETE

### Failure

- VK\_ERROR\_OUT\_OF\_HOST\_MEMORY
- VK\_ERROR\_OUT\_OF\_DEVICE\_MEMORY
- VK\_ERROR\_SURFACE\_LOST\_KHR

Possible values of elements of the `vkGetPhysicalDeviceSurfacePresentModesKHR::pPresentModes` array, indicating the supported presentation modes for a surface, are:

```
typedef enum VkPresentModeKHR {
    VK_PRESENT_MODE_IMMEDIATE_KHR = 0,
    VK_PRESENT_MODE_MAILBOX_KHR = 1,
    VK_PRESENT_MODE_FIFO_KHR = 2,
    VK_PRESENT_MODE_FIFO_RELAXED_KHR = 3,
    VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR = 1000111000,
    VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR = 1000111001,
    VK_PRESENT_MODE_MAX_ENUM_KHR = 0x7FFFFFFF
} VkPresentModeKHR;
```

- `VK_PRESENT_MODE_IMMEDIATE_KHR` specifies that the presentation engine does not wait for a vertical blanking period to update the current image, meaning this mode **may** result in visible tearing. No internal queuing of presentation requests is needed, as the requests are applied immediately.
- `VK_PRESENT_MODE_MAILBOX_KHR` specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing **cannot** be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for re-use by the application. One request is removed from the queue and processed during each vertical blanking period in which the queue is non-empty.
- `VK_PRESENT_MODE_FIFO_KHR` specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing **cannot** be observed. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty. This is the only value of `presentMode` that is **required** to be supported.
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR` specifies that the presentation engine generally waits for the next vertical blanking period to update the current image. If a vertical blanking period has already passed since the last update of the current image then the presentation engine does not wait for another vertical blanking period for the update, meaning this mode **may** result in

visible tearing in this case. This mode is useful for reducing visual stutter with an application that will mostly present a new image before the next vertical blanking period, but may occasionally be late, and present a new image just after the next vertical blanking period. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during or after each vertical blanking period in which the queue is non-empty.

- `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a *shared presentable image*. The presentation engine is only required to update the current image after a new presentation request is received. Therefore the application **must** make a presentation request whenever an update is required. However, the presentation engine **may** update the current image at any point, meaning this mode **may** result in visible tearing.
- `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a *shared presentable image*. The presentation engine periodically updates the current image on its regular refresh cycle. The application is only required to make one initial presentation request, after which the presentation engine **must** update the current image without any need for further presentation requests. The application **can** indicate the image contents have been updated by making a presentation request, but this does not guarantee the timing of when it will be updated. This mode **may** result in visible tearing if rendering to the image is not timed correctly.

The supported `VkImageUsageFlagBits` of the presentable images of a swapchain created for a surface **may** differ depending on the presentation mode, and can be determined as per the table below:

*Table 44. Presentable image usage queries*

Presentation mode	Image usage flags
<code>VK_PRESENT_MODE_IMMEDIATE_KHR</code>	<code>VkSurfaceCapabilitiesKHR::supportedUsageFlags</code>
<code>VK_PRESENT_MODE_MAILBOX_KHR</code>	<code>VkSurfaceCapabilitiesKHR::supportedUsageFlags</code>
<code>VK_PRESENT_MODE_FIFO_KHR</code>	<code>VkSurfaceCapabilitiesKHR::supportedUsageFlags</code>
<code>VK_PRESENT_MODE_FIFO_RELAXED_KHR</code>	<code>VkSurfaceCapabilitiesKHR::supportedUsageFlags</code>
<code>VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR</code>	<code>VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags</code>
<code>VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR</code>	<code>VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags</code>

#### Note

For reference, the mode indicated by `VK_PRESENT_MODE_FIFO_KHR` is equivalent to the behavior of {wgl|glX|egl}SwapBuffers with a swap interval of 1, while the mode indicated by `VK_PRESENT_MODE_FIFO_RELAXED_KHR` is equivalent to the behavior of {wgl|glX}SwapBuffers with a swap interval of -1 (from the {WGL|GLX}\_EXT\_swap\_control\_tear extensions).



## 32.6. Full Screen Exclusive Control

Swapchains created with `fullScreenExclusive` set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT` **must** acquire and release exclusive full-screen access explicitly, using the following commands.

To acquire exclusive full-screen access for a swapchain, call:

```
VkResult vkAcquireFullScreenExclusiveModeEXT(  
    VkDevice device,  
    VkSwapchainKHR swapchain);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to acquire exclusive full-screen access for.

### Valid Usage

- `swapchain` **must** not be in the retired state
- `swapchain` **must** be a swapchain created with an instance of `VkSurfaceFullScreenExclusiveInfoEXT`, with `fullScreenExclusive` set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`
- `swapchain` **must** not currently have exclusive full-screen access

A return value of `VK_SUCCESS` indicates that the `swapchain` successfully acquired exclusive full-screen access. The swapchain will retain this exclusivity until either the application releases exclusive full-screen access with `vkReleaseFullScreenExclusiveModeEXT`, destroys the swapchain, or if any of the swapchain commands return `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT` indicating that the mode was lost because of platform-specific changes.

If the swapchain was unable to acquire exclusive full-screen access to the display then `VK_ERROR_INITIALIZATION_FAILED` is returned. An application **can** attempt to acquire exclusive full-screen access again for the same swapchain even if this command fails, or if `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT` has been returned by a swapchain command.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- Both of `device`, and `swapchain` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`
- `VK_ERROR_SURFACE_LOST_KHR`

To release exclusive full-screen access from a swapchain, call:

```
VkResult vkReleaseFullScreenExclusiveModeEXT(  
    VkDevice device,  
    VkSwapchainKHR swapchain);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to release exclusive full-screen access from.

*Note*



Applications will not be able to present to `swapchain` after this call until exclusive full-screen access is reacquired. This is usually useful to handle when an application is minimised or otherwise intends to stop presenting for a time.

## Valid Usage

- `swapchain` **must** not be in the retired state
- `swapchain` **must** be a swapchain created with an instance of `VkSurfaceFullScreenExclusiveInfoEXT`, with `fullScreenExclusive` set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`

## 32.7. Device Group Queries

A logical device that represents multiple physical devices **may** support presenting from images on more than one physical device, or combining images from multiple physical devices.

To query these capabilities, call:

```
VkResult vkGetDeviceGroupPresentCapabilitiesKHR(  
    VkDevice device,  
    VkDeviceGroupPresentCapabilitiesKHR* pDeviceGroupPresentCapabilities);
```

- `device` is the logical device.
- `pDeviceGroupPresentCapabilities` is a pointer to a `VkDeviceGroupPresentCapabilitiesKHR` structure in which the device's capabilities are returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pDeviceGroupPresentCapabilities` **must** be a valid pointer to a `VkDeviceGroupPresentCapabilitiesKHR` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDeviceGroupPresentCapabilitiesKHR` structure is defined as:

```
typedef struct VkDeviceGroupPresentCapabilitiesKHR {
    VkStructureType           sType;
    const void*               pNext;
    uint32_t                  presentMask[VK_MAX_DEVICE_GROUP_SIZE];
    VkDeviceGroupPresentModeFlagsKHR modes;
} VkDeviceGroupPresentCapabilitiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `presentMask` is an array of `VK_MAX_DEVICE_GROUP_SIZE uint32_t` masks, where the mask at element `i` is non-zero if physical device `i` has a presentation engine, and where bit `j` is set in element `i` if physical device `i` **can** present swapchain images from physical device `j`. If element `i` is non-zero, then bit `i` **must** be set.
- `modes` is a bitmask of `VkDeviceGroupPresentModeFlagBitsKHR` indicating which device group presentation modes are supported.

`modes` always has `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR` set.

The present mode flags are also used when presenting an image, in `VkDeviceGroupPresentInfoKHR::mode`.

If a device group only includes a single physical device, then `modes` **must** equal `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR`
- `pNext` must be `NULL`

Bits which **may** be set in `VkDeviceGroupPresentCapabilitiesKHR::modes` to indicate which device group presentation modes are supported are:

```
typedef enum VkDeviceGroupPresentModeFlagBitsKHR {
    VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR = 0x00000001,
    VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR = 0x00000002,
    VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR = 0x00000004,
    VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR = 0x00000008,
    VK_DEVICE_GROUP_PRESENT_MODE_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkDeviceGroupPresentModeFlagBitsKHR;
```

- `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR` specifies that any physical device with a presentation engine **can** present its own swapchain images.
- `VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR` specifies that any physical device with a presentation engine **can** present swapchain images from any physical device in its `presentMask`.
- `VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR` specifies that any physical device with a presentation engine **can** present the sum of swapchain images from any physical devices in its `presentMask`.
- `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR` specifies that multiple physical devices with a presentation engine **can** each present their own swapchain images.

```
typedef VkFlags VkDeviceGroupPresentModeFlagsKHR;
```

`VkDeviceGroupPresentModeFlagsKHR` is a bitmask type for setting a mask of zero or more `VkDeviceGroupPresentModeFlagBitsKHR`.

Some surfaces **may** not be capable of using all the device group present modes.

To query the supported device group present modes for a particular surface, call:

```
VkResult vkGetDeviceGroupSurfacePresentModesKHR(
    VkDevice                                     device,
    VkSurfaceKHR                                  surface,
    VkDeviceGroupPresentModeFlagsKHR*            pModes);
```

- `device` is the logical device.
- `surface` is the surface.
- `pModes` is a pointer to a `VkDeviceGroupPresentModeFlagsKHR` in which the supported device group present modes for the surface are returned.

The modes returned by this command are not invariant, and **may** change in response to the surface being moved, resized, or occluded. These modes **must** be a subset of the modes returned by [vkGetDeviceGroupPresentCapabilitiesKHR](#).

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pModes` **must** be a valid pointer to a `VkDeviceGroupPresentModeFlagsKHR` value
- Both of `device`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Host Synchronization

- Host access to `surface` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

Alternatively, to query the supported device group presentation modes for a surface combined with select other fixed swapchain creation parameters, call:

```
VkResult vkGetDeviceGroupSurfacePresentModes2EXT(  
    VkDevice                                     device,  
    const VkPhysicalDeviceSurfaceInfo2KHR*       pCreateInfo,  
    VkDeviceGroupPresentModeFlagsKHR*            pModes);
```

- `device` is the logical device.
- `pCreateInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by [vkCreateSwapchainKHR](#).
- `pModes` is a pointer to a `VkDeviceGroupPresentModeFlagsKHR` in which the supported device group present modes for the surface are returned.

`vkGetDeviceGroupSurfacePresentModes2EXT` behaves similarly to `vkGetDeviceGroupSurfacePresentModesKHR`, with the ability to specify extended inputs via

chained input structures.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pSurfaceInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceSurfaceInfo2KHR` structure
- `pModes` **must** be a valid pointer to a `VkDeviceGroupPresentModeFlagsKHR` value

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

When using `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR`, the application **may** need to know which regions of the surface are used when presenting locally on each physical device. Presentation of swapchain images to this surface need only have valid contents in the regions returned by this command.

To query a set of rectangles used in presentation on the physical device, call:

```
VkResult vkGetPhysicalDevicePresentRectanglesKHR(  
    VkPhysicalDevice                physicalDevice,  
    VkSurfaceKHR                    surface,  
    uint32_t*                      pRectCount,  
    VkRect2D*                      pRects);
```

- `physicalDevice` is the physical device.
- `surface` is the surface.
- `pRectCount` is a pointer to an integer related to the number of rectangles available or queried, as described below.
- `pRects` is either `NULL` or a pointer to an array of `VkRect2D` structures.

If `pRects` is `NULL`, then the number of rectangles used when presenting the given `surface` is returned in `pRectCount`. Otherwise, `pRectCount` **must** point to a variable set by the user to the number of elements in the `pRects` array, and on return the variable is overwritten with the number of structures actually written to `pRects`. If the value of `pRectCount` is less than the number of rectangles, at most `pRectCount` structures will be written. If `pRectCount` is smaller than the number of rectangles used for the given `surface`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

The values returned by this command are not invariant, and **may** change in response to the surface being moved, resized, or occluded.

The rectangles returned by this command **must** not overlap.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pRectCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pRectCount` is not `0`, and `pRects` is not `NULL`, `pRects` **must** be a valid pointer to an array of `pRectCount` `VkRect2D` structures
- Both of `physicalDevice`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

### Host Synchronization

- Host access to `surface` **must** be externally synchronized

### Return Codes

#### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

#### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

## 32.8. Display Timing Queries

Traditional game and real-time-animation applications frequently use `VK_PRESENT_MODE_FIFO_KHR` so that presentable images are updated during the vertical blanking period of a given refresh cycle (RC) of the presentation engine's display. This avoids the visual anomaly known as tearing.

However, synchronizing the presentation of images with the RC does not prevent all forms of visual anomalies. Stuttering occurs when the geometry for each presentable image is not accurately positioned for when that image will be displayed. The geometry may appear to move too little some RCs, and too much for others. Sometimes the animation appears to freeze, when the same image is used for more than one RC.

In order to minimize stuttering, an application needs to correctly position their geometry for when the presentable image will be displayed to the user. To accomplish this, applications need various

timing information about the presentation engine's display. They need to know when presentable images were actually presented, and when they could have been presented. Applications also need to tell the presentation engine to display an image no sooner than a given time. This can allow the application's animation to look smooth to the user, with no stuttering. The `VK_GOOGLE_display_timing` extension allows an application to satisfy these needs.

The presentation engine's display typically refreshes the pixels that are displayed to the user on a periodic basis. The period may be fixed or variable. In many cases, the presentation engine is associated with fixed refresh rate (FRR) display technology, with a fixed refresh rate (RR, e.g. 60Hz). In some cases, the presentation engine is associated with variable refresh rate (VRR) display technology, where each refresh cycle (RC) can vary in length. This extension treats VRR displays as if they are FRR.

To query the duration of a refresh cycle (RC) for the presentation engine's display, call:

```
VkResult vkGetRefreshCycleDurationGOOGLE(  
    VkDevice device,  
    VkSwapchainKHR swapchain,  
    VkRefreshCycleDurationGOOGLE* pDisplayTimingProperties);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to obtain the refresh duration for.
- `pDisplayTimingProperties` is a pointer to a `VkRefreshCycleDurationGOOGLE` structure.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- `pDisplayTimingProperties` **must** be a valid pointer to a `VkRefreshCycleDurationGOOGLE` structure
- Both of `device`, and `swapchain` **must** have been created, allocated, or retrieved from the same `VkInstance`

### Host Synchronization

- Host access to `swapchain` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkRefreshCycleDurationGOOGLE` structure is defined as:

```
typedef struct VkRefreshCycleDurationGOOGLE {
    uint64_t    refreshDuration;
} VkRefreshCycleDurationGOOGLE;
```

- `refreshDuration` is the number of nanoseconds from the start of one refresh cycle to the next.

#### Note

The rate at which an application renders and presents new images is known as the image present rate (IPR, aka frame rate). The inverse of IPR, or the duration between each image present, is the image present duration (IPD). In order to provide a smooth, stutter-free animation, an application will want its IPD to be a multiple of `refreshDuration`. For example, if a display has a 60Hz refresh rate, `refreshDuration` will be a value in nanoseconds that is approximately equal to 16.67ms. In such a case, an application will want an IPD of 16.67ms (1X multiplier of `refreshDuration`), or 33.33ms (2X multiplier of `refreshDuration`), or 50.0ms (3X multiplier of `refreshDuration`), etc.

In order to determine a target IPD for a display (i.e. a multiple of `refreshDuration`), an application needs to determine when its images are actually displayed. Let's say that an application has an initial target IPD of 16.67ms (1X multiplier of `refreshDuration`). It will therefore position the geometry of a new image 16.67ms later than the previous image. Let's say that this application is running on slower hardware, so that it actually takes 20ms to render each new image. This will create visual anomalies, because the images will not be displayed to the user every 16.67ms, nor every 20ms. In this case, it is better for the application to adjust its target IPD to 33.33ms (i.e. a 2X multiplier of `refreshDuration`), and tell the presentation engine to not present images any sooner than every 33.33ms. This will allow the geometry to be correctly positioned for each presentable image.

Adjustments to an application's IPD may be needed because different views of an application's geometry can take different amounts of time to render. For example, looking at the sky may take less time to render than looking at multiple, complex items in a room. In general, it is good to not frequently change IPD, as that can cause visual anomalies. Adjustments to a larger IPD because of late images should happen quickly, but adjustments to a smaller IPD should only happen if the `actualPresentTime` and `earliestPresentTime` members of the `VkPastPresentationTimingGOOGLE` structure are consistently different, and if `presentMargin` is consistently large, over multiple images.

The implementation will maintain a limited amount of history of timing information about previous presents. Because of the asynchronous nature of the presentation engine, the timing information for a given `vkQueuePresentKHR` command will become available some time later. These time values can be asynchronously queried, and will be returned if available. All time values are in nanoseconds, relative to a monotonically-increasing clock (e.g. `CLOCK_MONOTONIC` (see `clock_gettime(2)`) on Android and Linux).

To asynchronously query the presentation engine, for newly-available timing information about one or more previous presents to a given swapchain, call:

```
VkResult vkGetPastPresentationTimingGOOGLE(  
    VkDevice  
    VkSwapchainKHR  
    uint32_t*  
    VkPastPresentationTimingGOOGLE*  
        device,  
        swapchain,  
        pPresentationTimingCount,  
        pPresentationTimings);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to obtain presentation timing information duration for.
- `pPresentationTimingCount` is a pointer to an integer related to the number of `VkPastPresentationTimingGOOGLE` structures to query, as described below.
- `pPresentationTimings` is either `NULL` or a pointer to an array of `VkPastPresentationTimingGOOGLE` structures.

If `pPresentationTimings` is `NULL`, then the number of newly-available timing records for the given `swapchain` is returned in `pPresentationTimingCount`. Otherwise, `pPresentationTimingCount` **must** point to a variable set by the user to the number of elements in the `pPresentationTimings` array, and on return the variable is overwritten with the number of structures actually written to `pPresentationTimings`. If the value of `pPresentationTimingCount` is less than the number of newly-available timing records, at most `pPresentationTimingCount` structures will be written. If `pPresentationTimingCount` is smaller than the number of newly-available timing records for the given `swapchain`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- `pPresentationTimingCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPresentationTimingCount` is not `0`, and `pPresentationTimings` is not `NULL`, `pPresentationTimings` **must** be a valid pointer to an array of `pPresentationTimingCount` `VkPastPresentationTimingGOOGLE` structures
- Both of `device`, and `swapchain` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Host Synchronization

- Host access to `swapchain` **must** be externally synchronized

## Return Codes

### Success

- VK\_SUCCESS
- VK\_INCOMPLETE

### Failure

- VK\_ERROR\_DEVICE\_LOST
- VK\_ERROR\_OUT\_OF\_DATE\_KHR
- VK\_ERROR\_SURFACE\_LOST\_KHR

The `VkPastPresentationTimingGOOGLE` structure is defined as:

```
typedef struct VkPastPresentationTimingGOOGLE {
    uint32_t    presentID;
    uint64_t    desiredPresentTime;
    uint64_t    actualPresentTime;
    uint64_t    earliestPresentTime;
    uint64_t    presentMargin;
} VkPastPresentationTimingGOOGLE;
```

- `presentID` is an application-provided value that was given to a previous `vkQueuePresentKHR` command via `VkPresentTimeGOOGLE::presentID` (see below). It **can** be used to uniquely identify a previous present with the `vkQueuePresentKHR` command.
- `desiredPresentTime` is an application-provided value that was given to a previous `vkQueuePresentKHR` command via `VkPresentTimeGOOGLE::desiredPresentTime`. If non-zero, it was used by the application to indicate that an image not be presented any sooner than `desiredPresentTime`.
- `actualPresentTime` is the time when the image of the `swapchain` was actually displayed.
- `earliestPresentTime` is the time when the image of the `swapchain` could have been displayed. This **may** differ from `actualPresentTime` if the application requested that the image be presented no sooner than `VkPresentTimeGOOGLE::desiredPresentTime`.
- `presentMargin` is an indication of how early the `vkQueuePresentKHR` command was processed compared to how soon it needed to be processed, and still be presented at `earliestPresentTime`.

The results for a given `swapchain` and `presentID` are only returned once from `vkGetPastPresentationTimingGOOGLE`.

The application **can** use the `VkPastPresentationTimingGOOGLE` values to occasionally adjust its timing. For example, if `actualPresentTime` is later than expected (e.g. one `refreshDuration` late), the application may increase its target IPD to a higher multiple of `refreshDuration` (e.g. decrease its frame rate from 60Hz to 30Hz). If `actualPresentTime` and `earliestPresentTime` are consistently different, and if `presentMargin` is consistently large enough, the application may decrease its target IPD to a smaller multiple of `refreshDuration` (e.g. increase its frame rate from 30Hz to 60Hz). If `actualPresentTime` and `earliestPresentTime` are same, and if `presentMargin` is consistently high, the

application may delay the start of its input-render-present loop in order to decrease the latency between user input and the corresponding present (always leaving some margin in case a new image takes longer to render than the previous image). An application that desires its target IPD to always be the same as `refreshDuration`, can also adjust features until `actualPresentTime` is never late and `presentMargin` is satisfactory.

The full `VK_GOOGLE_display_timing` extension semantics are described for swapchains created with `VK_PRESENT_MODE_FIFO_KHR`. For example, non-zero values of `VkPresentTimeGOOGLE::desiredPresentTime` **must** be honored, and `vkGetPastPresentationTimingGOOGLE` **should** return a `VkPastPresentationTimingGOOGLE` structure with valid values for all images presented with `vkQueuePresentKHR`. The semantics for other present modes are as follows:

- **`VK_PRESENT_MODE_IMMEDIATE_KHR`**. The presentation engine **may** ignore non-zero values of `VkPresentTimeGOOGLE::desiredPresentTime` in favor of presenting immediately. The value of `VkPastPresentationTimingGOOGLE::earliestPresentTime` **must** be the same as `VkPastPresentationTimingGOOGLE::actualPresentTime`, which **should** be when the presentation engine displayed the image.
- **`VK_PRESENT_MODE_MAILBOX_KHR`**. The intention of using this present mode with this extension is to handle cases where an image is presented late, and the next image is presented soon enough to replace it at the next vertical blanking period. For images that are displayed to the user, the value of `VkPastPresentationTimingGOOGLE::actualPresentTime` **must** be when the image was displayed. For images that are not displayed to the user, `vkGetPastPresentationTimingGOOGLE` **may** not return a `VkPastPresentationTimingGOOGLE` structure, or it **may** return a `VkPastPresentationTimingGOOGLE` structure with the value of zero for both `VkPastPresentationTimingGOOGLE::actualPresentTime` and `VkPastPresentationTimingGOOGLE::earliestPresentTime`. It is possible that an application **can** submit images with `VkPresentTimeGOOGLE::desiredPresentTime` values such that new images **may** not be displayed. For example, if `VkPresentTimeGOOGLE::desiredPresentTime` is far enough in the future that an image is not presented before `vkQueuePresentKHR` is called to present another image, the first image will not be displayed to the user. If the application continues to do that, the presentation **may** not display new images.
- **`VK_PRESENT_MODE_FIFO_RELAXED_KHR`**. For images that are presented in time to be displayed at the next vertical blanking period, the semantics are identical as for `VK_PRESENT_MODE_FIFO_RELAXED_KHR`. For images that are presented late, and are displayed after the start of the vertical blanking period (i.e. with tearing), the values of `VkPastPresentationTimingGOOGLE` **may** be treated as if the image was displayed at the start of the vertical blanking period, or **may** be treated the same as for `VK_PRESENT_MODE_IMMEDIATE_KHR`.

## 32.9. WSI Swapchain

A swapchain object (a.k.a. swapchain) provides the ability to present rendering results to a surface. Swapchain objects are represented by `VkSwapchainKHR` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSwapchainKHR)
```

A swapchain is an abstraction for an array of presentable images that are associated with a surface.

The presentable images are represented by `VkImage` objects created by the platform. One image (which **can** be an array image for multiview/stereoscopic-3D surfaces) is displayed at a time, but multiple images **can** be queued for presentation. An application renders to the image, and then queues the image for presentation to the surface.

A native window **cannot** be associated with more than one non-retired swapchain at a time. Further, swapchains **cannot** be created for native windows that have a non-Vulkan graphics API surface associated with them.

*Note*

The presentation engine is an abstraction for the platform's compositor or display engine.



The presentation engine **may** be synchronous or asynchronous with respect to the application and/or logical device.

Some implementations **may** use the device's graphics queue or dedicated presentation hardware to perform presentation.

The presentable images of a swapchain are owned by the presentation engine. An application **can** acquire use of a presentable image from the presentation engine. Use of a presentable image **must** occur only after the image is returned by `vkAcquireNextImageKHR`, and before it is presented by `vkQueuePresentKHR`. This includes transitioning the image layout and rendering commands.

An application **can** acquire use of a presentable image with `vkAcquireNextImageKHR`. After acquiring a presentable image and before modifying it, the application **must** use a synchronization primitive to ensure that the presentation engine has finished reading from the image. The application **can** then transition the image's layout, queue rendering commands to it, etc. Finally, the application presents the image with `vkQueuePresentKHR`, which releases the acquisition of the image.

The presentation engine controls the order in which presentable images are acquired for use by the application.

*Note*



This allows the platform to handle situations which require out-of-order return of images after presentation. At the same time, it allows the application to generate command buffers referencing all of the images in the swapchain at initialization time, rather than in its main loop.

How this all works is described below.

If a swapchain is created with `presentMode` set to either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, a single presentable image **can** be acquired, referred to as a shared presentable image. A shared presentable image **may** be concurrently accessed by the application and the presentation engine, without transitioning the image's layout after it is initially presented.

- With `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR`, the presentation engine is only required to update to the latest contents of a shared presentable image after a present. The application

**must** call `vkQueuePresentKHR` to guarantee an update. However, the presentation engine **may** update from it at any time.

- With `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, the presentation engine will automatically present the latest contents of a shared presentable image during every refresh cycle. The application is only required to make one initial call to `vkQueuePresentKHR`, after which the presentation engine will update from it without any need for further present calls. The application **can** indicate the image contents have been updated by calling `vkQueuePresentKHR`, but this does not guarantee the timing of when updates will occur.

The presentation engine **may** access a shared presentable image at any time after it is first presented. To avoid tearing, an application **should** coordinate access with the presentation engine. This requires presentation engine timing information through platform-specific mechanisms and ensuring that color attachment writes are made available during the portion of the presentation engine's refresh cycle they are intended for.

*Note*



The `VK_KHR_shared_presentable_image` extension does not provide functionality for determining the timing of the presentation engine's refresh cycles.

In order to query a swapchain's status when rendering to a shared presentable image, call:

```
VkResult vkGetSwapchainStatusKHR(  
    VkDevice                                     device,  
    VkSwapchainKHR                               swapchain);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to query.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- Both of `device`, and `swapchain` **must** have been created, allocated, or retrieved from the same `VkInstance`

### Host Synchronization

- Host access to `swapchain` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_SUBOPTIMAL_KHR`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_OUT_OF_DATE_KHR`
- `VK_ERROR_SURFACE_LOST_KHR`
- `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT`

The possible return values for `vkGetSwapchainStatusKHR` **should** be interpreted as follows:

- `VK_SUCCESS` specifies the presentation engine is presenting the contents of the shared presentable image, as per the swapchain's `VkPresentModeKHR`.
- `VK_SUBOPTIMAL_KHR` the swapchain no longer matches the surface properties exactly, but the presentation engine is presenting the contents of the shared presentable image, as per the swapchain's `VkPresentModeKHR`.
- `VK_ERROR_OUT_OF_DATE_KHR` the surface has changed in such a way that it is no longer compatible with the swapchain.
- `VK_ERROR_SURFACE_LOST_KHR` the surface is no longer available.

#### Note



The swapchain state **may** be cached by implementations, so applications **should** regularly call `vkGetSwapchainStatusKHR` when using a swapchain with `VkPresentModeKHR` set to `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`.

To create a swapchain, call:

```
VkResult vkCreateSwapchainKHR(  
    VkDevice                                     device,  
    const VkSwapchainCreateInfoKHR* pCreateInfo,  
    const VkAllocationCallbacks*     pAllocator,  
    VkSwapchainKHR*                  pSwapchain);
```

- `device` is the device to create the swapchain for.
- `pCreateInfo` is a pointer to a `VkSwapchainCreateInfoKHR` structure specifying the parameters of the created swapchain.
- `pAllocator` is the allocator used for host memory allocated for the swapchain object when there is no more specific allocator available (see [Memory Allocation](#)).

- `pSwapchain` is a pointer to a `VkSwapchainKHR` handle in which the created swapchain object will be returned.

If the `oldSwapchain` parameter of `pCreateInfo` is a valid swapchain, which has exclusive full-screen access, that access is released from `oldSwapchain`. If the command succeeds in this case, the newly created swapchain will automatically acquire exclusive full-screen access from `oldSwapchain`.

*Note*



This implicit transfer is intended to avoid exiting and entering full-screen exclusive mode, which may otherwise cause unwanted visual updates to the display.

In some cases, swapchain creation **may** fail if exclusive full-screen mode is requested for application control, but for some implementation-specific reason exclusive full-screen access is unavailable for the particular combination of parameters provided. If this occurs, `VK_ERROR_INITIALIZATION_FAILED` will be returned.

*Note*



In particular, it will fail if the `imageExtent` member of `pCreateInfo` does not match the extents of the monitor. Other reasons for failure may include the app not being set as high-dpi aware, or if the physical device and monitor are not compatible in this mode.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkSwapchainCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSwapchain` **must** be a valid pointer to a `VkSwapchainKHR` handle

## Host Synchronization

- Host access to `pCreateInfo.surface` **must** be externally synchronized
- Host access to `pCreateInfo.oldSwapchain` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_SURFACE_LOST_KHR`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`
- `VK_ERROR_INITIALIZATION_FAILED`

The `VkSwapchainCreateInfoKHR` structure is defined as:

```
typedef struct VkSwapchainCreateInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkSwapchainCreateFlagsKHR flags;
    VkSurfaceKHR               surface;
    uint32_t                  minImageCount;
    VkFormat                   imageFormat;
    VkColorSpaceKHR            imageColorSpace;
    VkExtent2D                 imageExtent;
    uint32_t                  imageArrayLayers;
    VkImageUsageFlags          imageUsage;
    VkSharingMode               imageSharingMode;
    uint32_t                  queueFamilyIndexCount;
    const uint32_t*             pQueueFamilyIndices;
    VkSurfaceTransformFlagBitsKHR preTransform;
    VkCompositeAlphaFlagBitsKHR compositeAlpha;
    VkPresentModeKHR            presentMode;
    VkBool32                   clipped;
    VkSwapchainKHR              oldSwapchain;
} VkSwapchainCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkSwapchainCreateFlagBitsKHR` indicating parameters of the swapchain creation.
- `surface` is the surface onto which the swapchain will present images. If the creation succeeds, the swapchain becomes associated with `surface`.
- `minImageCount` is the minimum number of presentable images that the application needs. The implementation will either create the swapchain with at least that many images, or it will fail to create the swapchain.

- `imageFormat` is a [VkFormat](#) value specifying the format the swapchain image(s) will be created with.
- `imageColorSpace` is a [VkColorSpaceKHR](#) value specifying the way the swapchain interprets image data.
- `imageExtent` is the size (in pixels) of the swapchain image(s). The behavior is platform-dependent if the image extent does not match the surface's `currentExtent` as returned by [vkGetPhysicalDeviceSurfaceCapabilitiesKHR](#).

*Note*



On some platforms, it is normal that `maxImageExtent` **may** become `(0, 0)`, for example when the window is minimized. In such a case, it is not possible to create a swapchain due to the Valid Usage requirements.

- `imageArrayLayers` is the number of views in a multiview/stereo surface. For non-stereoscopic-3D applications, this value is 1.
- `imageUsage` is a bitmask of [VkImageUsageFlagBits](#) describing the intended usage of the (acquired) swapchain images.
- `imageSharingMode` is the sharing mode used for the image(s) of the swapchain.
- `queueFamilyIndexCount` is the number of queue families having access to the image(s) of the swapchain when `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`.
- `pQueueFamilyIndices` is a pointer to an array of queue family indices having access to the images(s) of the swapchain when `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`.
- `preTransform` is a [VkSurfaceTransformFlagBitsKHR](#) value describing the transform, relative to the presentation engine's natural orientation, applied to the image content prior to presentation. If it does not match the `currentTransform` value returned by [vkGetPhysicalDeviceSurfaceCapabilitiesKHR](#), the presentation engine will transform the image content as part of the presentation operation.
- `compositeAlpha` is a [VkCompositeAlphaFlagBitsKHR](#) value indicating the alpha compositing mode to use when this surface is composited together with other surfaces on certain window systems.
- `presentMode` is the presentation mode the swapchain will use. A swapchain's present mode determines how incoming present requests will be processed and queued internally.
- `clipped` specifies whether the Vulkan implementation is allowed to discard rendering operations that affect regions of the surface that are not visible.
  - If set to `VK_TRUE`, the presentable images associated with the swapchain **may** not own all of their pixels. Pixels in the presentable images that correspond to regions of the target surface obscured by another window on the desktop, or subject to some other clipping mechanism will have undefined content when read back. Fragment shaders **may** not execute for these pixels, and thus any side effects they would have had will not occur. `VK_TRUE` value does not guarantee any clipping will occur, but allows more optimal presentation methods to be used on some platforms.
  - If set to `VK_FALSE`, presentable images associated with the swapchain will own all of the pixels they contain.

*Note*



Applications **should** set this value to `VK_TRUE` if they do not expect to read back the content of presentable images before presenting them or after reacquiring them, and if their fragment shaders do not have any side effects that require them to run for all pixels in the presentable image.

- `oldSwapchain` is `VK_NULL_HANDLE`, or the existing non-retired swapchain currently associated with `surface`. Providing a valid `oldSwapchain` **may** aid in the resource reuse, and also allows the application to still present any images that are already acquired from it.

Upon calling `vkCreateSwapchainKHR` with an `oldSwapchain` that is not `VK_NULL_HANDLE`, `oldSwapchain` is retired—even if creation of the new swapchain fails. The new swapchain is created in the non-retired state whether or not `oldSwapchain` is `VK_NULL_HANDLE`.

Upon calling `vkCreateSwapchainKHR` with an `oldSwapchain` that is not `VK_NULL_HANDLE`, any images from `oldSwapchain` that are not acquired by the application **may** be freed by the implementation, which **may** occur even if creation of the new swapchain fails. The application **can** destroy `oldSwapchain` to free all memory associated with `oldSwapchain`.

*Note*

Multiple retired swapchains **can** be associated with the same `VkSurfaceKHR` through multiple uses of `oldSwapchain` that outnumber calls to `vkDestroySwapchainKHR`.



After `oldSwapchain` is retired, the application **can** pass to `vkQueuePresentKHR` any images it had already acquired from `oldSwapchain`. E.g., an application may present an image from the old swapchain before an image from the new swapchain is ready to be presented. As usual, `vkQueuePresentKHR` **may** fail if `oldSwapchain` has entered a state that causes `VK_ERROR_OUT_OF_DATE_KHR` to be returned.

The application **can** continue to use a shared presentable image obtained from `oldSwapchain` until a presentable image is acquired from the new swapchain, as long as it has not entered a state that causes it to return `VK_ERROR_OUT_OF_DATE_KHR`.

## Valid Usage

- `surface` **must** be a surface that is supported by the device as determined using `vkGetPhysicalDeviceSurfaceSupportKHR`
- `minImageCount` **must** be greater than or equal to the value returned in the `minImageCount` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface
- `minImageCount` **must** be less than or equal to the value returned in the `maxImageCount` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface if the returned `maxImageCount` is not zero
- `minImageCount` **must** be 1 if `presentMode` is either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`
- `imageFormat` and `imageColorSpace` **must** match the `format` and `colorSpace` members, respectively, of one of the `VkSurfaceFormatKHR` structures returned by `vkGetPhysicalDeviceSurfaceFormatsKHR` for the surface
- `imageExtent` **must** be between `minImageExtent` and `maxImageExtent`, inclusive, where `minImageExtent` and `maxImageExtent` are members of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface
- `imageExtent` members `width` and `height` **must** both be non-zero
- `imageArrayLayers` **must** be greater than 0 and less than or equal to the `maxImageArrayLayers` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface
- If `presentMode` is `VK_PRESENT_MODE_IMMEDIATE_KHR`, `VK_PRESENT_MODE_MAILBOX_KHR`, `VK_PRESENT_MODE_FIFO_KHR` or `VK_PRESENT_MODE_FIFO_RELAXED_KHR`, `imageUsage` **must** be a subset of the supported usage flags present in the `supportedUsageFlags` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for `surface`
- If `presentMode` is `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, `imageUsage` **must** be a subset of the supported usage flags present in the `sharedPresentSupportedUsageFlags` member of the `VkSharedPresentSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilities2KHR` for `surface`
- If `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` **must** be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values
- If `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` **must** be greater than 1
- If `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` **must** be unique and **must** be less than `pQueueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create `device`

- `preTransform` **must** be one of the bits present in the `supportedTransforms` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface
- `compositeAlpha` **must** be one of the bits present in the `supportedCompositeAlpha` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface
- `presentMode` **must** be one of the `VkPresentModeKHR` values returned by `vkGetPhysicalDeviceSurfacePresentModesKHR` for the surface
- If the logical device was created with `VkDeviceGroupDeviceCreateInfo`  
`::physicalDeviceCount` equal to 1, `flags` **must** not contain `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`
- If `oldSwapchain` is not `VK_NULL_HANDLE`, `oldSwapchain` **must** be a non-retired swapchain associated with native window referred to by `surface`
- The `implied image creation parameters` of the swapchain **must** be supported as reported by `vkGetPhysicalDeviceImageFormatProperties`
- If `flags` contains `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR` then the `pNext` chain **must** contain an instance of `VkImageFormatListCreateInfoKHR` with a `viewFormatCount` greater than zero and `pViewFormats` **must** have an element equal to `imageFormat`
- If `flags` contains `VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR`, then `VkSurfaceProtectedCapabilitiesKHR::supportsProtected` **must** be `VK_TRUE` in the `VkSurfaceProtectedCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilities2KHR` for `surface`
- If the `pNext` chain includes an instance of `VkSurfaceFullScreenExclusiveInfoEXT` with its `fullScreenExclusive` member set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`, and `surface` was created using `vkCreateWin32SurfaceKHR`, an instance of `VkSurfaceFullScreenExclusiveWin32InfoEXT` **must** be present in the `pNext` chain

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkDeviceGroupSwapchainCreateInfoKHR`, `VkImageFormatListCreateInfoKHR`, `VkSurfaceFullScreenExclusiveInfoEXT`, `VkSurfaceFullScreenExclusiveWin32InfoEXT`, `VkSwapchainCounterCreateInfoEXT`, or `VkSwapchainDisplayNativeHdrCreateInfoAMD`
- Each `sType` member in the `pNext` chain **must** be unique
- `flags` **must** be a valid combination of `VkSwapchainCreateFlagBitsKHR` values
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `imageFormat` **must** be a valid `VkFormat` value
- `imageColorSpace` **must** be a valid `VkColorSpaceKHR` value
- `imageUsage` **must** be a valid combination of `VkImageUsageFlagBits` values
- `imageUsage` **must** not be `0`
- `imageSharingMode` **must** be a valid `VkSharingMode` value
- `preTransform` **must** be a valid `VkSurfaceTransformFlagBitsKHR` value
- `compositeAlpha` **must** be a valid `VkCompositeAlphaFlagBitsKHR` value
- `presentMode` **must** be a valid `VkPresentModeKHR` value
- If `oldSwapchain` is not `VK_NULL_HANDLE`, `oldSwapchain` **must** be a valid `VkSwapchainKHR` handle
- If `oldSwapchain` is a valid handle, it **must** have been created, allocated, or retrieved from `surface`
- Both of `oldSwapchain`, and `surface` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkInstance`

Bits which **can** be set in `VkSwapchainCreateInfoKHR::flags`, specifying parameters of swapchain creation, are:

```
typedef enum VkSwapchainCreateFlagBitsKHR {
    VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR = 0x00000001,
    VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR = 0x00000002,
    VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR = 0x00000004,
    VK_SWAPCHAIN_CREATE_FLAG_BITS_MAX_ENUM_KHR = 0x7FFFFFFF
} VkSwapchainCreateFlagBitsKHR;
```

- `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR` specifies that images created from the swapchain (i.e. with the `swapchain` member of `VkImageSwapchainCreateInfoKHR` set to this swapchain's handle) **must** use `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`.
- `VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR` specifies that images created from the swapchain are

protected images.

- `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR` specifies that the images of the swapchain **can** be used to create a `VkImageView` with a different format than what the swapchain was created with. The list of allowed image view formats are specified by chaining an instance of the `VkImageFormatListCreateInfoKHR` structure to the `pNext` chain of `VkSwapchainCreateInfoKHR`. In addition, this flag also specifies that the swapchain **can** be created with usage flags that are not supported for the format the swapchain is created with but are supported for at least one of the allowed image view formats.

```
typedef VkFlags VkSwapchainCreateFlagsKHR;
```

`VkSwapchainCreateFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSwapchainCreateFlagBitsKHR`.

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkDeviceGroupSwapchainCreateInfoKHR` structure, then that structure includes a set of device group present modes that the swapchain **can** be used with.

The `VkDeviceGroupSwapchainCreateInfoKHR` structure is defined as:

```
typedef struct VkDeviceGroupSwapchainCreateInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    VkDeviceGroupPresentModeFlagsKHR modes;
} VkDeviceGroupSwapchainCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `modes` is a bitfield of modes that the swapchain **can** be used with.

If this structure is not present, `modes` is considered to be `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`
- `modes` **must** be a valid combination of `VkDeviceGroupPresentModeFlagBitsKHR` values
- `modes` **must** not be `0`

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkSwapchainDisplayNativeHdrCreateInfoAMD` structure, then that structure includes additional swapchain creation parameters specific to display native HDR support.

The `VkSwapchainDisplayNativeHdrCreateInfoAMD` structure is defined as:

```
typedef struct VkSwapchainDisplayNativeHdrCreateInfoAMD {
    VkStructureType sType;
    const void* pNext;
    VkBool32 localDimmingEnable;
} VkSwapchainDisplayNativeHdrCreateInfoAMD;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `localDimmingEnable` specifies whether local dimming is enabled for the swapchain.

If the `pNext` chain of `VkSwapchainCreateInfoKHR` does not contain this structure, the default value for `localDimmingEnable` is `VK_TRUE`, meaning local dimming is initially enabled for the swapchain.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SWAPCHAIN_DISPLAY_NATIVE_HDR_CREATE_INFO_AMD`

### Valid Usage

- It is only valid to set `localDimmingEnable` to `VK_TRUE` if `VkDisplayNativeHdrSurfaceCapabilitiesAMD::localDimmingSupport` is supported.

The local dimming HDR setting may also be changed over the life of a swapchain by calling:

```
void vkSetLocalDimmingAMD(
    VkDevice device,
    VkSwapchainKHR swapChain,
    VkBool32 localDimmingEnable);
```

- `device` is the device associated with `swapChain`.
- `swapChain` handle to enable local dimming.
- `localDimmingEnable` specifies whether local dimming is enabled for the swapchain.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapChain` **must** be a valid `VkSwapchainKHR` handle
- Both of `device`, and `swapChain` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Valid Usage

- It is only valid to call `vkSetLocalDimmingAMD` if `VkDisplayNativeHdrSurfaceCapabilitiesAMD::localDimmingSupport` is supported.

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkSurfaceFullScreenExclusiveInfoEXT` structure, then that structure specifies the application's preferred full-screen presentation behavior. If this structure is not present, `fullScreenExclusive` is considered to be `VK_FULL_SCREEN_EXCLUSIVE_DEFAULT_EXT`.

To enable surface counters when creating a swapchain, add `VkSwapchainCounterCreateInfoEXT` to the `pNext` chain of `VkSwapchainCreateInfoKHR`. `VkSwapchainCounterCreateInfoEXT` is defined as:

```
typedef struct VkSwapchainCounterCreateInfoEXT {
    VkStructureType          sType;
    const void*              pNext;
    VkSurfaceCounterFlagsEXT surfaceCounters;
} VkSwapchainCounterCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `surfaceCounters` is a bitmask of `VkSurfaceCounterFlagBitsEXT` specifying surface counters to enable for the swapchain.

## Valid Usage

- The bits in `surfaceCounters` **must** be supported by `VkSwapchainCreateInfoKHR::surface`, as reported by `vkGetPhysicalDeviceSurfaceCapabilities2EXT`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SWAPCHAIN_COUNTER_CREATE_INFO_EXT`
- `surfaceCounters` **must** be a valid combination of `VkSurfaceCounterFlagBitsEXT` values

The requested counters become active when the first presentation command for the associated swapchain is processed by the presentation engine. To query the value of an active counter, use:

```
VkResult vkGetSwapchainCounterEXT(
    VkDevice                      device,
    VkSwapchainKHR                swapchain,
    VkSurfaceCounterFlagBitsEXT   counter,
    uint64_t*                     pCounterValue);
```

- `device` is the `VkDevice` associated with `swapchain`.
- `swapchain` is the swapchain from which to query the counter value.
- `counter` is the counter to query.
- `pCounterValue` will return the current value of the counter.

If a counter is not available because the swapchain is out of date, the implementation **may** return `VK_ERROR_OUT_OF_DATE_KHR`.

## Valid Usage

- One or more present commands on `swapchain` **must** have been processed by the presentation engine.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- `counter` **must** be a valid `VkSurfaceCounterFlagBitsEXT` value
- `pCounterValue` **must** be a valid pointer to a `uint64_t` value
- Both of `device`, and `swapchain` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_OUT_OF_DATE_KHR`

As mentioned above, if `vkCreateSwapchainKHR` succeeds, it will return a handle to a swapchain containing an array of at least `minImageCount` presentable images.

While acquired by the application, presentable images **can** be used in any way that equivalent non-presentable images **can** be used. A presentable image is equivalent to a non-presentable image created with the following `VkImageCreateInfo` parameters:

VkImageCreateInfo Field	Value
flags	<p><code>VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT</code> is set if  <code>VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR</code> is set</p> <p><code>VK_IMAGE_CREATE_PROTECTED_BIT</code> is set if  <code>VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR</code> is set</p> <p><code>VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT</code> and  <code>VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR</code> are both set if  <code>VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR</code> is set</p> <p>all other bits are unset</p>
imageType	<code>VK_IMAGE_TYPE_2D</code>
format	<code>pCreateInfo-&gt;imageFormat</code>
extent	<code>{pCreateInfo-&gt;imageExtent.width, pCreateInfo-&gt;imageExtent.height, 1}</code>
mipLevels	1
arrayLayers	<code>pCreateInfo-&gt;imageArrayLayers</code>
samples	<code>VK_SAMPLE_COUNT_1_BIT</code>
tiling	<code>VK_IMAGE_TILING_OPTIMAL</code>
usage	<code>pCreateInfo-&gt;imageUsage</code>
sharingMode	<code>pCreateInfo-&gt;imageSharingMode</code>
queueFamilyIndexCount	<code>pCreateInfo-&gt;queueFamilyIndexCount</code>
pQueueFamilyIndices	<code>pCreateInfo-&gt;pQueueFamilyIndices</code>
initialLayout	<code>VK_IMAGE_LAYOUT_UNDEFINED</code>

The `surface` **must** not be destroyed until after the swapchain is destroyed.

If `oldSwapchain` is `VK_NULL_HANDLE`, and the native window referred to by `surface` is already associated with a Vulkan swapchain, `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR` **must** be returned.

If the native window referred to by `surface` is already associated with a non-Vulkan graphics API surface, `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR` **must** be returned.

The native window referred to by `surface` **must** not become associated with a non-Vulkan graphics API surface before all associated Vulkan swapchains have been destroyed.

Like core functions, several WSI functions, including `vkCreateSwapchainKHR` return `VK_ERROR_DEVICE_LOST` if the logical device was lost. See [Lost Device](#). As with most core objects, `VkSwapchainKHR` is a child of the device and is affected by the lost state; it **must** be destroyed before destroying the `VkDevice`. However, `VkSurfaceKHR` is not a child of any `VkDevice` and is not otherwise affected by the lost device. After successfully recreating a `VkDevice`, the same `VkSurfaceKHR` **can** be used to create a new `VkSwapchainKHR`, provided the previous one was destroyed.

### Note



As mentioned in [Lost Device](#), after a lost device event, the `VkPhysicalDevice` **may** also be lost. If other `VkPhysicalDevice` are available, they **can** be used together with the same `VkSurfaceKHR` to create the new `VkSwapchainKHR`, however the application **must** query the surface capabilities again, because they **may** differ on a per-physical device basis.

To destroy a swapchain object call:

```
void vkDestroySwapchainKHR(  
    VkDevice device,  
    VkSwapchainKHR swapchain,  
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the `VkDevice` associated with `swapchain`.
- `swapchain` is the swapchain to destroy.
- `pAllocator` is the allocator used for host memory allocated for the swapchain object when there is no more specific allocator available (see [Memory Allocation](#)).

The application **must** not destroy a swapchain until after completion of all outstanding operations on images that were acquired from the swapchain. `swapchain` and all associated `VkImage` handles are destroyed, and **must** not be acquired or used any more by the application. The memory of each `VkImage` will only be freed after that image is no longer used by the presentation engine. For example, if one image of the swapchain is being displayed in a window, the memory for that image **may** not be freed until the window is destroyed, or another swapchain is created for the window. Destroying the swapchain does not invalidate the parent `VkSurfaceKHR`, and a new swapchain **can** be created with it.

When a swapchain associated with a display surface is destroyed, if the image most recently presented to the display surface is from the swapchain being destroyed, then either any display resources modified by presenting images from any swapchain associated with the display surface **must** be reverted by the implementation to their state prior to the first present performed on one of these swapchains, or such resources **must** be left in their current state.

If `swapchain` has exclusive full-screen access, it is released before the swapchain is destroyed.

### Valid Usage

- All uses of presentable images acquired from `swapchain` **must** have completed execution
- If `VkAllocationCallbacks` were provided when `swapchain` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `swapchain` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- If `swapchain` is not `VK_NULL_HANDLE`, `swapchain` **must** be a valid `VkSwapchainKHR` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- Both of `device`, and `swapchain` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkInstance`

## Host Synchronization

- Host access to `swapchain` **must** be externally synchronized

When the `VK_KHR_display_swapchain` extension is enabled, multiple swapchains that share presentable images are created by calling:

```
VkResult vkCreateSharedSwapchainsKHR(  
    VkDevice                                     device,  
    uint32_t                                      swapchainCount,  
    const VkSwapchainCreateInfoKHR*                pCreateInfos,  
    const VkAllocationCallbacks*                  pAllocator,  
    VkSwapchainKHR*                            pSwapchains);
```

- `device` is the device to create the swapchains for.
- `swapchainCount` is the number of swapchains to create.
- `pCreateInfos` is a pointer to an array of `VkSwapchainCreateInfoKHR` structures specifying the parameters of the created swapchains.
- `pAllocator` is the allocator used for host memory allocated for the swapchain objects when there is no more specific allocator available (see [Memory Allocation](#)).
- `pSwapchains` is a pointer to an array of `VkSwapchainKHR` handles in which the created swapchain objects will be returned.

`vkCreateSharedSwapchainsKHR` is similar to `vkCreateSwapchainKHR`, except that it takes an array of `VkSwapchainCreateInfoKHR` structures, and returns an array of swapchain objects.

The swapchain creation parameters that affect the properties and number of presentable images **must** match between all the swapchains. If the displays used by any of the swapchains do not use the same presentable image layout or are incompatible in a way that prevents sharing images, swapchain creation will fail with the result code `VK_ERROR_INCOMPATIBLE_DISPLAY_KHR`. If any error occurs, no swapchains will be created. Images presented to multiple swapchains **must** be re-acquired from all of them before transitioning away from `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`. After destroying one or more of the swapchains, the remaining swapchains and the presentable images **can** continue to be used.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfos` **must** be a valid pointer to an array of `swapchainCount` valid `VkSwapchainCreateInfoKHR` structures
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pSwapchains` **must** be a valid pointer to an array of `swapchainCount` `VkSwapchainKHR` handles
- `swapchainCount` **must** be greater than `0`

## Host Synchronization

- Host access to `pCreateInfos[]`.`surface` **must** be externally synchronized
- Host access to `pCreateInfos[]`.`oldSwapchain` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INCOMPATIBLE_DISPLAY_KHR`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_SURFACE_LOST_KHR`

To obtain the array of presentable images associated with a swapchain, call:

```
VkResult vkGetSwapchainImagesKHR(  
    VkDevice device,  
    VkSwapchainKHR swapchain,  
    uint32_t* pSwapchainImageCount,  
    VkImage* pSwapchainImages);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to query.
- `pSwapchainImageCount` is a pointer to an integer related to the number of presentable images available or queried, as described below.
- `pSwapchainImages` is either `NULL` or a pointer to an array of `VkImage` handles.

If `pSwapchainImages` is `NULL`, then the number of presentable images for `swapchain` is returned in `pSwapchainImageCount`. Otherwise, `pSwapchainImageCount` **must** point to a variable set by the user to the number of elements in the `pSwapchainImages` array, and on return the variable is overwritten with the number of structures actually written to `pSwapchainImages`. If the value of `pSwapchainImageCount` is less than the number of presentable images for `swapchain`, at most `pSwapchainImageCount` structures will be written. If `pSwapchainImageCount` is smaller than the number of presentable images for `swapchain`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- `pSwapchainImageCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pSwapchainImageCount` is not `0`, and `pSwapchainImages` is not `NULL`, `pSwapchainImages` **must** be a valid pointer to an array of `pSwapchainImageCount` `VkImage` handles
- Both of `device`, and `swapchain` **must** have been created, allocated, or retrieved from the same `VkInstance`

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

### Note



By knowing all presentable images used in the swapchain, the application **can** create command buffers that reference these images prior to entering its main rendering loop.

Images returned by `vkGetSwapchainImagesKHR` are fully backed by memory before they are passed to the application. All presentable images are initially in the `VK_IMAGE_LAYOUT_UNDEFINED` layout, thus before using presentable images, the application **must** transition them to a valid layout for the intended use.

Further, the lifetime of presentable images is controlled by the implementation, so applications **must** not destroy a presentable image. See `vkDestroySwapchainKHR` for further details on the lifetime of presentable images.

Images **can** also be created by using `vkCreateImage` with `VkImageSwapchainCreateInfoKHR` and bound to swapchain memory using `vkBindImageMemory2KHR` with `VkBindImageMemorySwapchainInfoKHR`. These images **can** be used anywhere swapchain images are used, and are useful in logical devices with multiple physical devices to create peer memory bindings of swapchain memory. These images and bindings have no effect on what memory is presented. Unlike images retrieved from `vkGetSwapchainImagesKHR`, these images **must** be destroyed with `vkDestroyImage`.

To acquire an available presentable image to use, and retrieve the index of that image, call:

```
VkResult vkAcquireNextImageKHR(  
    VkDevice device,  
    VkSwapchainKHR swapchain,  
    uint64_t timeout,  
    VkSemaphore semaphore,  
    VkFence fence,  
    uint32_t* pImageIndex);
```

- **device** is the device associated with `swapchain`.
- **swapchain** is the non-retired swapchain from which an image is being acquired.
- **timeout** specifies how long the function waits, in nanoseconds, if no image is available.
- **semaphore** is `VK_NULL_HANDLE` or a semaphore to signal.
- **fence** is `VK_NULL_HANDLE` or a fence to signal.
- **pImageIndex** is a pointer to a `uint32_t` in which the index of the next image to use (i.e. an index into the array of images returned by `vkGetSwapchainImagesKHR`) is returned.

## Valid Usage

- `swapchain` **must** not be in the retired state
- If `semaphore` is not `VK_NULL_HANDLE` it **must** be unsignaled
- If `semaphore` is not `VK_NULL_HANDLE` it **must** not have any uncompleted signal or wait operations pending
- If `fence` is not `VK_NULL_HANDLE` it **must** be unsignaled and **must** not be associated with any other queue command that has not yet completed execution on that queue
- `semaphore` and `fence` **must** not both be equal to `VK_NULL_HANDLE`
- If the number of currently acquired images is greater than the difference between the number of images in `swapchain` and the value of `VkSurfaceCapabilitiesKHR::minImageCount` as returned by a call to `vkGetPhysicalDeviceSurfaceCapabilities2KHR` with the `surface` used to create `swapchain`, `timeout` **must** not be `UINT64_MAX`
- `semaphore` **must** have a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- If `semaphore` is not `VK_NULL_HANDLE`, `semaphore` **must** be a valid `VkSemaphore` handle
- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be a valid `VkFence` handle
- `pImageIndex` **must** be a valid pointer to a `uint32_t` value
- If `semaphore` is a valid handle, it **must** have been created, allocated, or retrieved from `device`
- If `fence` is a valid handle, it **must** have been created, allocated, or retrieved from `device`
- Both of `device`, and `swapchain` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkInstance`

## Host Synchronization

- Host access to `swapchain` **must** be externally synchronized
- Host access to `semaphore` **must** be externally synchronized
- Host access to `fence` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_TIMEOUT`
- `VK_NOT_READY`
- `VK_SUBOPTIMAL_KHR`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_OUT_OF_DATE_KHR`
- `VK_ERROR_SURFACE_LOST_KHR`
- `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT`

When successful, `vkAcquireNextImageKHR` acquires a presentable image from `swapchain` that an application **can** use, and sets `pImageIndex` to the index of that image within the swapchain. The presentation engine **may** not have finished reading from the image at the time it is acquired, so the application **must** use `semaphore` and/or `fence` to ensure that the image layout and contents are not modified until the presentation engine reads have completed. If `semaphore` is not

`VK_NULL_HANDLE`, the application may assume that, once `vkAcquireNextImageKHR` returns, the semaphore signal operation referenced by `semaphore` has been submitted for execution. The order in which images are acquired is implementation-dependent, and **may** be different than the order the images were presented.

If `timeout` is zero, then `vkAcquireNextImageKHR` does not wait, and will either successfully acquire an image, or fail and return `VK_NOT_READY` if no image is available.

If the specified timeout period expires before an image is acquired, `vkAcquireNextImageKHR` returns `VK_TIMEOUT`. If `timeout` is `UINT64_MAX`, the timeout period is treated as infinite, and `vkAcquireNextImageKHR` will block until an image is acquired or an error occurs.

An image will eventually be acquired if the number of images that the application has currently acquired (but not yet presented) is less than or equal to the difference between the number of images in `swapchain` and the value of `VkSurfaceCapabilitiesKHR::minImageCount`. If the number of currently acquired images is greater than this, `vkAcquireNextImageKHR` **should** not be called; if it is, `timeout` **must** not be `UINT64_MAX`.

If an image is acquired successfully, `vkAcquireNextImageKHR` **must** either return `VK_SUCCESS`, or `VK_SUBOPTIMAL_KHR` if the swapchain no longer matches the surface properties exactly, but **can** still be used for presentation.

*Note*



This **may** happen, for example, if the platform surface has been resized but the platform is able to scale the presented images to the new size to produce valid surface updates. It is up to the application to decide whether it prefers to continue using the current swapchain in this state, or to re-create the swapchain to better match the platform surface properties.

If the swapchain images no longer match native surface properties, either `VK_SUBOPTIMAL_KHR` or `VK_ERROR_OUT_OF_DATE_KHR` **must** be returned. If `VK_ERROR_OUT_OF_DATE_KHR` is returned, no image is acquired and attempts to present previously acquired images to the swapchain will also fail with `VK_ERROR_OUT_OF_DATE_KHR`. Applications need to create a new swapchain for the surface to continue presenting if `VK_ERROR_OUT_OF_DATE_KHR` is returned.

If device loss occurs (see [Lost Device](#)) before the timeout has expired, `vkAcquireNextImageKHR` **must** return in finite time with either one of the allowed success codes, or `VK_ERROR_DEVICE_LOST`.

If `semaphore` is not `VK_NULL_HANDLE`, the semaphore **must** be unsignaled, with no signal or wait operations pending. It will become signaled when the application **can** use the image.

*Note*



Use of `semaphore` allows rendering operations to be recorded and submitted before the presentation engine has completed its use of the image.

If `fence` is not equal to `VK_NULL_HANDLE`, the fence **must** be unsignaled, with no signal operations pending. It will become signaled when the application **can** use the image.

*Note*

Applications **should** not rely on `vkAcquireNextImageKHR` blocking in order to meter their rendering speed. The implementation **may** return from this function immediately regardless of how many presentation requests are queued, and regardless of when queued presentation requests will complete relative to the call. Instead, applications **can** use `fence` to meter their frame generation work to match the presentation rate.



An application **must** wait until either the `semaphore` or `fence` is signaled before accessing the image's data.

*Note*

When the presentable image will be accessed by some stage S, the recommended idiom for ensuring correct synchronization is:



- The `VkSubmitInfo` used to submit the image layout transition for execution includes `vkAcquireNextImageKHR::semaphore` in its `pWaitSemaphores` member, with the corresponding element of `pWaitDstStageMask` including S.
- The `synchronization command` that performs any necessary image layout transition includes S in both the `srcStageMask` and `dstStageMask`.

After a successful return, the image indicated by `pImageIndex` and its data will be unmodified compared to when it was presented.

*Note*

Exclusive ownership of presentable images corresponding to a swapchain created with `VK_SHARING_MODE_EXCLUSIVE` as defined in [Resource Sharing](#) is not altered by a call to `vkAcquireNextImageKHR`. That means upon the first acquisition from such a swapchain presentable images are not owned by any queue family, while at subsequent acquisitions the presentable images remain owned by the queue family the image was previously presented on.



The possible return values for `vkAcquireNextImageKHR` depend on the `timeout` provided:

- `VK_SUCCESS` is returned if an image became available.
- `VK_ERROR_SURFACE_LOST_KHR` if the surface becomes no longer available.
- `VK_NOT_READY` is returned if `timeout` is zero and no image was available.
- `VK_TIMEOUT` is returned if `timeout` is greater than zero and less than `UINT64_MAX`, and no image became available within the time allowed.
- `VK_SUBOPTIMAL_KHR` is returned if an image became available, and the swapchain no longer matches the surface properties exactly, but **can** still be used to present to the surface successfully.

*Note*



This **may** happen, for example, if the platform surface has been resized but the platform is able to scale the presented images to the new size to produce valid surface updates. It is up to the application to decide whether it prefers to continue using the current swapchain indefinitely or temporarily in this state, or to re-create the swapchain to better match the platform surface properties.

- `VK_ERROR_OUT_OF_DATE_KHR` is returned if the surface has changed in such a way that it is no longer compatible with the swapchain, and further presentation requests using the swapchain will fail. Applications **must** query the new surface properties and recreate their swapchain if they wish to continue presenting to the surface.

If the native surface and presented image sizes no longer match, presentation **may** fail. If presentation does succeed, the mapping from the presented image to the native surface is implementation-defined. It is the application's responsibility to detect surface size changes and react appropriately. If presentation fails because of a mismatch in the surface and presented image sizes, a `VK_ERROR_OUT_OF_DATE_KHR` error will be returned.

*Note*



For example, consider a 4x3 window/surface that gets resized to be 3x4 (taller than wider). On some window systems, the portion of the window/surface that was previously and still is visible (the 3x3 part) will contain the same contents as before, while the remaining parts of the window will have undefined contents. Other window systems **may** squash/stretch the image to fill the new window size without any undefined contents, or apply some other mapping.

To acquire an available presentable image to use, and retrieve the index of that image, call:

```
VkResult vkAcquireNextImage2KHR(  
    VkDevice device,  
    const VkAcquireNextImageInfoKHR* pAcquireInfo,  
    uint32_t* pImageIndex);
```

- `device` is the device associated with `swapchain`.
- `pAcquireInfo` is a pointer to a `VkAcquireNextImageInfoKHR` structure containing parameters of the acquire.
- `pImageIndex` is a pointer to a `uint32_t` that is set to the index of the next image to use.

### Valid Usage

- If the number of currently acquired images is greater than the difference between the number of images in the `swapchain` member of `pAcquireInfo` and the value of `VkSurfaceCapabilitiesKHR::minImageCount` as returned by a call to `vkGetPhysicalDeviceSurfaceCapabilities2KHR` with the `surface` used to create `swapchain`, the `timeout` member of `pAcquireInfo` **must** not be `UINT64_MAX`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pAcquireInfo` **must** be a valid pointer to a valid `VkAcquireNextImageInfoKHR` structure
- `pImageIndex` **must** be a valid pointer to a `uint32_t` value

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_TIMEOUT`
- `VK_NOT_READY`
- `VK_SUBOPTIMAL_KHR`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_OUT_OF_DATE_KHR`
- `VK_ERROR_SURFACE_LOST_KHR`
- `VK_ERROR_FULLSCREEN_EXCLUSIVE_MODE_LOST_EXT`

The `VkAcquireNextImageInfoKHR` structure is defined as:

```
typedef struct VkAcquireNextImageInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
    uint64_t           timeout;
    VkSemaphore         semaphore;
    VkFence             fence;
    uint32_t            deviceMask;
} VkAcquireNextImageInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `swapchain` is a non-retired swapchain from which an image is acquired.
- `timeout` specifies how long the function waits, in nanoseconds, if no image is available.
- `semaphore` is `VK_NULL_HANDLE` or a semaphore to signal.
- `fence` is `VK_NULL_HANDLE` or a fence to signal.
- `deviceMask` is a mask of physical devices for which the swapchain image will be ready to use

when the semaphore or fence is signaled.

If [vkAcquireNextImageKHR](#) is used, the device mask is considered to include all physical devices in the logical device.

*Note*

[vkAcquireNextImage2KHR](#) signals at most one semaphore, even if the application requests waiting for multiple physical devices to be ready via the `deviceMask`. However, only a single physical device **can** wait on that semaphore, since the semaphore becomes unsignaled when the wait succeeds. For other physical devices to wait for the image to be ready, it is necessary for the application to submit semaphore signal operation(s) to that first physical device to signal additional semaphore(s) after the wait succeeds, which the other physical device(s) **can** wait upon.



## Valid Usage

- `swapchain` **must** not be in the retired state
- If `semaphore` is not `VK_NULL_HANDLE` it **must** be unsignaled
- If `semaphore` is not `VK_NULL_HANDLE` it **must** not have any uncompleted signal or wait operations pending
- If `fence` is not `VK_NULL_HANDLE` it **must** be unsignaled and **must** not be associated with any other queue command that has not yet completed execution on that queue
- `semaphore` and `fence` **must** not both be equal to `VK_NULL_HANDLE`
- `deviceMask` **must** be a valid device mask
- `deviceMask` **must** not be zero
- `semaphore` **must** have a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR`
- `pNext` **must** be `NULL`
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- If `semaphore` is not `VK_NULL_HANDLE`, `semaphore` **must** be a valid `VkSemaphore` handle
- If `fence` is not `VK_NULL_HANDLE`, `fence` **must** be a valid `VkFence` handle
- Each of `fence`, `semaphore`, and `swapchain` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkInstance`

## Host Synchronization

- Host access to `swapchain` **must** be externally synchronized
- Host access to `semaphore` **must** be externally synchronized
- Host access to `fence` **must** be externally synchronized

After queueing all rendering commands and transitioning the image to the correct layout, to queue an image for presentation, call:

```
VkResult vkQueuePresentKHR(  
    VkQueue           queue,  
    const VkPresentInfoKHR* pPresentInfo);
```

- `queue` is a queue that is capable of presentation to the target surface's platform on the same device as the image's swapchain.
- `pPresentInfo` is a pointer to a `VkPresentInfoKHR` structure specifying parameters of the presentation.

*Note*



There is no requirement for an application to present images in the same order that they were acquired - applications can arbitrarily present any image that is currently acquired.

## Valid Usage

- Each element of `pSwapchains` member of `pPresentInfo` **must** be a swapchain that is created for a surface for which presentation is supported from `queue` as determined using a call to `vkGetPhysicalDeviceSurfaceSupportKHR`
- If more than one member of `pSwapchains` was created from a display surface, all display surfaces referenced that refer to the same display **must** use the same display mode
- When a semaphore wait operation referring to a binary semaphore defined by the elements of the `pWaitSemaphores` member of `pPresentInfo` executes on `queue`, there **must** be no other queues waiting on the same semaphore.
- All elements of the `pWaitSemaphores` member of `pPresentInfo` **must** be semaphores that are signaled, or have `semaphore signal operations` previously submitted for execution.
- All elements of the `pWaitSemaphores` member of `pPresentInfo` **must** be created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR`.
- All elements of the `pWaitSemaphores` member of `pPresentInfo` **must** reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) **must** have also been submitted for execution.

Any writes to memory backing the images referenced by the `pImageIndices` and `pSwapchains`

members of `pPresentInfo`, that are available before `vkQueuePresentKHR` is executed, are automatically made visible to the read access performed by the presentation engine. This automatic visibility operation for an image happens-after the semaphore signal operation, and happens-before the presentation engine accesses the image.

Queueing an image for presentation defines a set of *queue operations*, including waiting on the semaphores and submitting a presentation request to the presentation engine. However, the scope of this set of queue operations does not include the actual processing of the image by the presentation engine.

If `vkQueuePresentKHR` fails to enqueue the corresponding set of queue operations, it **may** return `VK_ERROR_OUT_OF_HOST_MEMORY` or `VK_ERROR_OUT_OF_DEVICE_MEMORY`. If it does, the implementation **must** ensure that the state and contents of any resources or synchronization primitives referenced is unaffected by the call or its failure.

If `vkQueuePresentKHR` fails in such a way that the implementation is unable to make that guarantee, the implementation **must** return `VK_ERROR_DEVICE_LOST`.

However, if the presentation request is rejected by the presentation engine with an error `VK_ERROR_OUT_OF_DATE_KHR` or `VK_ERROR_SURFACE_LOST_KHR`, the set of queue operations are still considered to be enqueued and thus any semaphore wait operation specified in `VkPresentInfoKHR` will execute when the corresponding queue operation is complete.

If any `swapchain` member of `pPresentInfo` was created with `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`, `VK_ERROR_FULLSCREEN_EXCLUSIVE_MODE_LOST_EXT` will be returned if that swapchain does not have exclusive full-screen access, possibly for implementation-specific reasons outside of the application's control.

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- `pPresentInfo` **must** be a valid pointer to a valid `VkPresentInfoKHR` structure

## Host Synchronization

- Host access to `queue` **must** be externally synchronized
- Host access to `pPresentInfo.pWaitSemaphores[]` **must** be externally synchronized
- Host access to `pPresentInfo.pSwapchains[]` **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	Any	-

## Return Codes

### Success

- VK\_SUCCESS
- VK\_SUBOPTIMAL\_KHR

### Failure

- VK\_ERROR\_OUT\_OF\_HOST\_MEMORY
- VK\_ERROR\_OUT\_OF\_DEVICE\_MEMORY
- VK\_ERROR\_DEVICE\_LOST
- VK\_ERROR\_OUT\_OF\_DATE\_KHR
- VK\_ERROR\_SURFACE\_LOST\_KHR
- VK\_ERROR\_FULL\_SCREEN\_EXCLUSIVE\_MODE\_LOST\_EXT

The `VkPresentInfoKHR` structure is defined as:

```
typedef struct VkPresentInfoKHR {
    VkStructureType          sType;
    const void*               pNext;
    uint32_t                  waitSemaphoreCount;
    const VkSemaphore*        pWaitSemaphores;
    uint32_t                  swapchainCount;
    const VkSwapchainKHR*    pSwapchains;
    const uint32_t*           pImageIndices;
    VkResult*                 pResults;
} VkPresentInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `waitSemaphoreCount` is the number of semaphores to wait for before issuing the present request. The number **may** be zero.
- `pWaitSemaphores` is `NULL` or a pointer to an array of `VkSemaphore` objects with `waitSemaphoreCount` entries, and specifies the semaphores to wait for before issuing the present request.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pSwapchains` is a pointer to an array of `VkSwapchainKHR` objects with `swapchainCount` entries. A given swapchain **must** not appear in this list more than once.

- `pImageIndices` is a pointer to an array of indices into the array of each swapchain's presentable images, with `swapchainCount` entries. Each entry in this array identifies the image to present on the corresponding entry in the `pSwapchains` array.
- `pResults` is a pointer to an array of `VkResult` typed elements with `swapchainCount` entries. Applications that do not need per-swapchain results **can** use `NULL` for `pResults`. If non-`NULL`, each entry in `pResults` will be set to the `VkResult` for presenting the swapchain corresponding to the same index in `pSwapchains`.

Before an application **can** present an image, the image's layout **must** be transitioned to the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout, or for a shared presentable image the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` layout.

*Note*



When transitioning the image to `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` or `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`, there is no need to delay subsequent processing, or perform any visibility operations (as `vkQueuePresentKHR` performs automatic visibility operations). To achieve this, the `dstAccessMask` member of the `VkImageMemoryBarrier` **should** be set to `0`, and the `dstStageMask` parameter **should** be set to `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`.

## Valid Usage

- Each element of `pImageIndices` **must** be the index of a presentable image acquired from the swapchain specified by the corresponding element of the `pSwapchains` array, and the presented image subresource **must** be in the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` layout at the time the operation is executed on a `VkDevice`
- All elements of the `pWaitSemaphores` **must** have a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_BINARY_KHR`

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PRESENT_INFO_KHR`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDeviceGroupPresentInfoKHR`, `VkDisplayPresentInfoKHR`, `VkPresentFrameTokenGGP`, `VkPresentRegionsKHR`, or `VkPresentTimesInfoGOOGLE`
- Each `sType` member in the `pNext` chain must be unique
- If `waitSemaphoreCount` is not `0`, `pWaitSemaphores` must be a valid pointer to an array of `waitSemaphoreCount` valid `VkSemaphore` handles
- `pSwapchains` must be a valid pointer to an array of `swapchainCount` valid `VkSwapchainKHR` handles
- `pImageIndices` must be a valid pointer to an array of `swapchainCount uint32_t` values
- If `pResults` is not `NULL`, `pResults` must be a valid pointer to an array of `swapchainCount VkResult` values
- `swapchainCount` must be greater than `0`
- Both of the elements of `pSwapchains`, and the elements of `pWaitSemaphores` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkInstance`

When the `VK_KHR_incremental_present` extension is enabled, additional fields can be specified that allow an application to specify that only certain rectangular regions of the presentable images of a swapchain are changed. This is an optimization hint that a presentation engine may use to only update the region of a surface that is actually changing. The application still must ensure that all pixels of a presented image contain the desired values, in case the presentation engine ignores this hint. An application can provide this hint by including the `VkPresentRegionsKHR` structure in the `pNext` chain of the `VkPresentInfoKHR` structure.

The `VkPresentRegionsKHR` structure is defined as:

```
typedef struct VkPresentRegionsKHR {
    VkStructureType          sType;
    const void*               pNext;
    uint32_t                  swapchainCount;
    const VkPresentRegionKHR* pRegions;
} VkPresentRegionsKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pRegions` is `NULL` or a pointer to an array of `VkPresentRegionKHR` elements with `swapchainCount` entries. If not `NULL`, each element of `pRegions` contains the region that has changed since the last

present to the swapchain in the corresponding entry in the `VkPresentInfoKHR::pSwapchains` array.

## Valid Usage

- `swapchainCount` **must** be the same value as `VkPresentInfoKHR::swapchainCount`, where `VkPresentInfoKHR` is in the `pNext` chain of this `VkPresentRegionsKHR` structure

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR`
- If `pRegions` is not `NULL`, `pRegions` **must** be a valid pointer to an array of `swapchainCount` valid `VkPresentRegionKHR` structures
- `swapchainCount` **must** be greater than `0`

For a given image and swapchain, the region to present is specified by the `VkPresentRegionKHR` structure, which is defined as:

```
typedef struct VkPresentRegionKHR {
    uint32_t          rectangleCount;
    const VkRectLayerKHR* pRectangles;
} VkPresentRegionKHR;
```

- `rectangleCount` is the number of rectangles in `pRectangles`, or zero if the entire image has changed and should be presented.
- `pRectangles` is either `NULL` or a pointer to an array of `VkRectLayerKHR` structures. The `VkRectLayerKHR` structure is the framebuffer coordinates, plus layer, of a portion of a presentable image that has changed and **must** be presented. If non-`NULL`, each entry in `pRectangles` is a rectangle of the given image that has changed since the last image was presented to the given swapchain.

## Valid Usage (Implicit)

- If `rectangleCount` is not `0`, and `pRectangles` is not `NULL`, `pRectangles` **must** be a valid pointer to an array of `rectangleCount` valid `VkRectLayerKHR` structures

The `VkRectLayerKHR` structure is defined as:

```
typedef struct VkRectLayerKHR {
    VkOffset2D    offset;
    VkExtent2D    extent;
    uint32_t      layer;
} VkRectLayerKHR;
```

- `offset` is the origin of the rectangle, in pixels.
- `extent` is the size of the rectangle, in pixels.
- `layer` is the layer of the image. For images with only one layer, the value of `layer` **must** be 0.

## Valid Usage

- The sum of `offset` and `extent` **must** be no greater than the `imageExtent` member of the `VkSwapchainCreateInfoKHR` structure given to `vkCreateSwapchainKHR`.
- `layer` **must** be less than `imageArrayLayers` member of the `VkSwapchainCreateInfoKHR` structure given to `vkCreateSwapchainKHR`.

Some platforms allow the size of a surface to change, and then scale the pixels of the image to fit the surface. `VkRectLayerKHR` specifies pixels of the swapchain's image(s), which will be constant for the life of the swapchain.

When the `VK_KHR_display_swapchain` extension is enabled additional fields **can** be specified when presenting an image to a swapchain by setting `VkPresentInfoKHR::pNext` to point to an instance of the `VkDisplayPresentInfoKHR` structure.

The `VkDisplayPresentInfoKHR` structure is defined as:

```
typedef struct VkDisplayPresentInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkRect2D           srcRect;
    VkRect2D           dstRect;
    VkBool32           persistent;
} VkDisplayPresentInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcRect` is a rectangular region of pixels to present. It **must** be a subset of the image being presented. If `VkDisplayPresentInfoKHR` is not specified, this region will be assumed to be the entire presentable image.
- `dstRect` is a rectangular region within the visible region of the swapchain's display mode. If `VkDisplayPresentInfoKHR` is not specified, this region will be assumed to be the entire visible region of the visible region of the swapchain's mode. If the specified rectangle is a subset of the display mode's visible region, content from display planes below the swapchain's plane will be visible outside the rectangle. If there are no planes below the swapchain's, the area outside the specified rectangle will be black. If portions of the specified rectangle are outside of the display's visible region, pixels mapping only to those portions of the rectangle will be discarded.
- `persistent`: If this is `VK_TRUE`, the display engine will enable buffered mode on displays that support it. This allows the display engine to stop sending content to the display until a new image is presented. The display will instead maintain a copy of the last presented image. This

allows less power to be used, but **may** increase presentation latency. If [VkDisplayPresentInfoKHR](#) is not specified, persistent mode will not be used.

If the extent of the `srcRect` and `dstRect` are not equal, the presented pixels will be scaled accordingly.

## Valid Usage

- `srcRect` **must** specify a rectangular region that is a subset of the image being presented
- `dstRect` **must** specify a rectangular region that is a subset of the `visibleRegion` parameter of the display mode the swapchain being presented uses
- If the `persistentContent` member of the [VkDisplayPropertiesKHR](#) structure returned by `vkGetPhysicalDeviceDisplayPropertiesKHR` for the display the present operation targets then `persistent` **must** be `VK_FALSE`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR`

If the `pNext` chain of [VkPresentInfoKHR](#) includes a [VkDeviceGroupPresentInfoKHR](#) structure, then that structure includes an array of device masks and a device group present mode.

The [VkDeviceGroupPresentInfoKHR](#) structure is defined as:

```
typedef struct VkDeviceGroupPresentInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    uint32_t                  swapchainCount;
    const uint32_t*            pDeviceMasks;
    VkDeviceGroupPresentModeFlagBitsKHR mode;
} VkDeviceGroupPresentInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `swapchainCount` is zero or the number of elements in `pDeviceMasks`.
- `pDeviceMasks` is a pointer to an array of device masks, one for each element of `VkPresentInfoKHR::pSwapchains`.
- `mode` is the device group present mode that will be used for this present.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`, then each element of `pDeviceMasks` selects which instance of the swapchain image is presented. Each element of `pDeviceMasks` **must** have exactly one bit set, and the corresponding physical device **must** have a presentation engine as reported by [VkDeviceGroupPresentCapabilitiesKHR](#).

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR`, then each element of `pDeviceMasks` selects which instance of the swapchain image is presented. Each element of `pDeviceMasks` **must** have exactly one bit set, and some physical device in the logical device **must** include that bit in its `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR`, then each element of `pDeviceMasks` selects which instances of the swapchain image are component-wise summed and the sum of those images is presented. If the sum in any component is outside the representable range, the value of that component is undefined. Each element of `pDeviceMasks` **must** have a value for which all set bits are set in one of the elements of `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR`, then each element of `pDeviceMasks` selects which instance(s) of the swapchain images are presented. For each bit set in each element of `pDeviceMasks`, the corresponding physical device **must** have a presentation engine as reported by `VkDeviceGroupPresentCapabilitiesKHR`.

If `VkDeviceGroupPresentInfoKHR` is not provided or `swapchainCount` is zero then the masks are considered to be 1. If `VkDeviceGroupPresentInfoKHR` is not provided, `mode` is considered to be `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`.

## Valid Usage

- `swapchainCount` **must** equal 0 or `VkPresentInfoKHR::swapchainCount`
- If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`, then each element of `pDeviceMasks` **must** have exactly one bit set, and the corresponding element of `VkDeviceGroupPresentCapabilitiesKHR::presentMask` **must** be non-zero
- If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR`, then each element of `pDeviceMasks` **must** have exactly one bit set, and some physical device in the logical device **must** include that bit in its `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.
- If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR`, then each element of `pDeviceMasks` **must** have a value for which all set bits are set in one of the elements of `VkDeviceGroupPresentCapabilitiesKHR::presentMask`
- If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR`, then for each bit set in each element of `pDeviceMasks`, the corresponding element of `VkDeviceGroupPresentCapabilitiesKHR::presentMask` **must** be non-zero
- The value of each element of `pDeviceMasks` **must** be equal to the device mask passed in `VkAcquireNextImageInfoKHR::deviceMask` when the image index was last acquired
- `mode` **must** have exactly one bit set, and that bit **must** have been included in `VkDeviceGroupSwapchainCreateInfoKHR::modes`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR`
- If `swapchainCount` is not `0`, `pDeviceMasks` **must** be a valid pointer to an array of `swapchainCount uint32_t` values
- `mode` **must** be a valid `VkDeviceGroupPresentModeFlagBitsKHR` value

When the `VK_GOOGLE_display_timing` extension is enabled, additional fields **can** be specified that allow an application to specify the earliest time that an image should be displayed. This allows an application to avoid stutter that is caused by an image being displayed earlier than planned. Such stuttering can occur with both fixed and variable-refresh-rate displays, because stuttering occurs when the geometry is not correctly positioned for when the image is displayed. An application **can** instruct the presentation engine that an image should not be displayed earlier than a specified time by including the `VkPresentTimesInfoGOOGLE` structure in the `pNext` chain of the `VkPresentInfoKHR` structure.

The `VkPresentTimesInfoGOOGLE` structure is defined as:

```
typedef struct VkPresentTimesInfoGOOGLE {
    VkStructureType           sType;
    const void*               pNext;
    uint32_t                  swapchainCount;
    const VkPresentTimeGOOGLE* pTimes;
} VkPresentTimesInfoGOOGLE;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pTimes` is `NULL` or a pointer to an array of `VkPresentTimeGOOGLE` elements with `swapchainCount` entries. If not `NULL`, each element of `pTimes` contains the earliest time to present the image corresponding to the entry in the `VkPresentInfoKHR::pImageIndices` array.

## Valid Usage

- `swapchainCount` **must** be the same value as `VkPresentInfoKHR::swapchainCount`, where `VkPresentInfoKHR` is in the `pNext` chain of this `VkPresentTimesInfoGOOGLE` structure.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PRESENT_TIMES_INFO_GOOGLE`
- If `pTimes` is not `NULL`, `pTimes` must be a valid pointer to an array of `swapchainCount` `VkPresentTimeGOOGLE` structures
- `swapchainCount` must be greater than `0`

The `VkPresentTimeGOOGLE` structure is defined as:

```
typedef struct VkPresentTimeGOOGLE {
    uint32_t    presentID;
    uint64_t    desiredPresentTime;
} VkPresentTimeGOOGLE;
```

- `presentID` is an application-provided identification value, that can be used with the results of `vkGetPastPresentationTimingGOOGLE`, in order to uniquely identify this present. In order to be useful to the application, it should be unique within some period of time that is meaningful to the application.
- `desiredPresentTime` specifies that the image given should not be displayed to the user any earlier than this time. `desiredPresentTime` is a time in nanoseconds, relative to a monotonically-increasing clock (e.g. `CLOCK_MONOTONIC` (see `clock_gettime(2)`) on Android and Linux). A value of zero specifies that the presentation engine may display the image at any time. This is useful when the application desires to provide `presentID`, but does not need a specific `desiredPresentTime`.

When the `VK_GGP_frame_token` extension is enabled, a Google Games Platform frame token can be specified when presenting an image to a swapchain by including the `VkPresentFrameTokenGGP` structure in the `pNext` chain of the `VkPresentInfoKHR` structure.

The `VkPresentFrameTokenGGP` structure is defined as:

```
typedef struct VkPresentFrameTokenGGP {
    VkStructureType    sType;
    const void*        pNext;
    GgpFrameToken     frameToken;
} VkPresentFrameTokenGGP;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `frameToken` is the Google Games Platform frame token.

## Valid Usage

- `frameToken` **must** be a valid `GgpFrameToken`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PRESENT_FRAME_TOKEN_GGP`

`vkQueuePresentKHR`, releases the acquisition of the images referenced by `imageIndices`. The queue family corresponding to the queue `vkQueuePresentKHR` is executed on **must** have ownership of the presented images as defined in [Resource Sharing](#). `vkQueuePresentKHR` does not alter the queue family ownership, but the presented images **must** not be used again before they have been reacquired using `vkAcquireNextImageKHR`.

The processing of the presentation happens in issue order with other queue operations, but semaphores have to be used to ensure that prior rendering and other commands in the specified queue complete before the presentation begins. The presentation command itself does not delay processing of subsequent commands on the queue, however, presentation requests sent to a particular queue are always performed in order. Exact presentation timing is controlled by the semantics of the presentation engine and native platform in use.

If an image is presented to a swapchain created from a display surface, the mode of the associated display will be updated, if necessary, to match the mode specified when creating the display surface. The mode switch and presentation of the specified image will be performed as one atomic operation.

The result codes `VK_ERROR_OUT_OF_DATE_KHR` and `VK_SUBOPTIMAL_KHR` have the same meaning when returned by `vkQueuePresentKHR` as they do when returned by `vkAcquireNextImageKHR`. If multiple swapchains are presented, the result code is determined applying the following rules in order:

- If the device is lost, `VK_ERROR_DEVICE_LOST` is returned.
- If any of the target surfaces are no longer available the error `VK_ERROR_SURFACE_LOST_KHR` is returned.
- If any of the presents would have a result of `VK_ERROR_OUT_OF_DATE_KHR` if issued separately then `VK_ERROR_OUT_OF_DATE_KHR` is returned.
- If any of the presents would have a result of `VK_ERROR_FULLSCREEN_EXCLUSIVE_MODE_LOST_EXT` if issued separately then `VK_ERROR_FULLSCREEN_EXCLUSIVE_MODE_LOST_EXT` is returned.
- If any of the presents would have a result of `VK_SUBOPTIMAL_KHR` if issued separately then `VK_SUBOPTIMAL_KHR` is returned.
- Otherwise `VK_SUCCESS` is returned.

Presentation is a read-only operation that will not affect the content of the presentable images. Upon reacquiring the image and transitioning it away from the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout, the contents will be the same as they were prior to transitioning the image to the present source layout and presenting it. However, if a mechanism other than Vulkan is used to modify the

platform window associated with the swapchain, the content of all presentable images in the swapchain becomes undefined.

*Note*



The application **can** continue to present any acquired images from a retired swapchain as long as the swapchain has not entered a state that causes `vkQueuePresentKHR` to return `VK_ERROR_OUT_OF_DATE_KHR`.

## 32.10. Hdr Metadata

To improve color reproduction of content it is useful to have information that can be used to better reproduce the colors as seen on the mastering display. That information can be provided to an implementation by calling `vkSetHdrMetadataEXT`. The metadata will be applied to the specified `VkSwapchainKHR` objects at the next `vkQueuePresentKHR` call using that `VkSwapchainKHR` object. The metadata will persist until a subsequent `vkSetHdrMetadataEXT` changes it. The definitions below are from the associated SMPTE 2086, CTA 861.3 and CIE 15:2004 specifications.

The definition of `vkSetHdrMetadataEXT` is:

```
void vkSetHdrMetadataEXT(  
    VkDevice                                     device,  
    uint32_t                                     swapchainCount,  
    const VkSwapchainKHR*                         pSwapchains,  
    const VkHdrMetadataEXT*                      pMetadata);
```

- `device` is the logical device where the swapchain(s) were created.
- `swapchainCount` is the number of swapchains included in `pSwapchains`.
- `pSwapchains` is a pointer to an array of `swapchainCount` `VkSwapchainKHR` handles.
- `pMetadata` is a pointer to an array of `swapchainCount` `VkHdrMetadataEXT` structures.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pSwapchains` **must** be a valid pointer to an array of `swapchainCount` valid `VkSwapchainKHR` handles
- `pMetadata` **must** be a valid pointer to an array of `swapchainCount` valid `VkHdrMetadataEXT` structures
- `swapchainCount` **must** be greater than 0
- Both of `device`, and the elements of `pSwapchains` **must** have been created, allocated, or retrieved from the same `VkInstance`

```
typedef struct VkXYColorEXT {
    float    x;
    float    y;
} VkXYColorEXT;
```

Chromaticity coordinates `x` and `y` are as specified in CIE 15:2004 “Calculation of chromaticity coordinates” (Section 7.3) and are limited to between 0 and 1 for real colors for the mastering display.

```
typedef struct VkHdrMetadataEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkXYColorEXT       displayPrimaryRed;
    VkXYColorEXT       displayPrimaryGreen;
    VkXYColorEXT       displayPrimaryBlue;
    VkXYColorEXT       whitePoint;
    float              maxLuminance;
    float              minLuminance;
    float              maxContentLightLevel;
    float              maxFrameAverageLightLevel;
} VkHdrMetadataEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `displayPrimaryRed` is the mastering display’s red primary in chromaticity coordinates
- `displayPrimaryGreen` is the mastering display’s green primary in chromaticity coordinates
- `displayPrimaryBlue` is the mastering display’s blue primary in chromaticity coordinates
- `whitePoint` is the mastering display’s white-point in chromaticity coordinates
- `maxLuminance` is the maximum luminance of the mastering display in nits
- `minLuminance` is the minimum luminance of the mastering display in nits
- `maxContentLightLevel` is content’s maximum luminance in nits
- `maxFrameAverageLightLevel` is the maximum frame average light level in nits

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_HDR_METADATA_EXT`
- `pNext` **must** be `NULL`



### Note

The validity and use of this data is outside the scope of Vulkan.

# Chapter 33. Ray Tracing

Unlike draw commands, which use rasterization, ray tracing is a rendering method that generates an image by tracing the path of rays which have a single origin and using shaders to determine the final colour of an image plane.

Ray tracing uses a separate rendering pipeline from both the graphics and compute pipelines (see [Ray tracing Pipeline](#)). It has a unique set of programmable and fixed function stages.

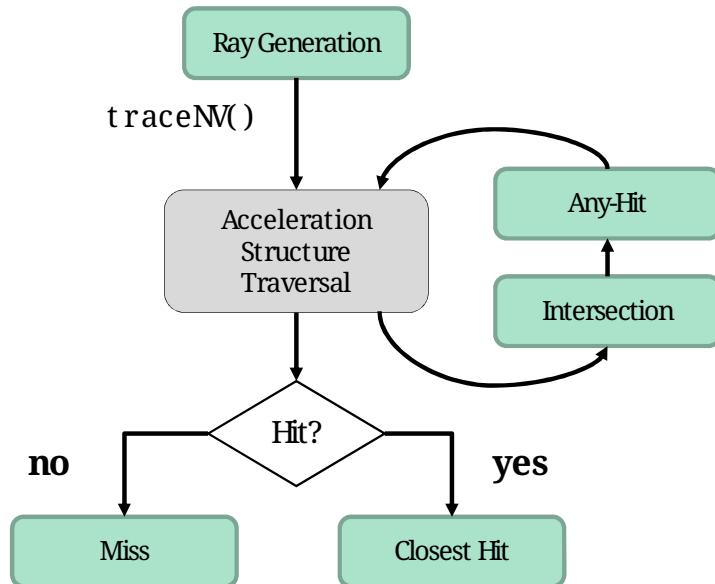


Figure 25. Ray tracing Pipeline

## Caption

Interaction between the different shader stages in the ray tracing pipeline

## 33.1. Ray Tracing Commands

*Ray tracing commands* provoke work in the ray tracing pipeline. Ray tracing commands are recorded into a command buffer and when executed by a queue will produce work that executes according to the currently bound ray tracing pipeline. A ray tracing pipeline **must** be bound to a command buffer before any ray tracing commands are recorded in that command buffer.

Each ray tracing call operates on a set of shader stages that are specific to the ray tracing pipeline as well as a set of `VkAccelerationStructureNV` objects describing the scene geometry in an implementation-specific way. The relationship between the ray tracing pipeline object and the acceleration structures is passed into the ray tracing command in a `VkBuffer` object known as a *shader binding table*.

During execution, control alternates between scheduling and other operations. The scheduling functionality is implementation-specific and is responsible for workload execution. The shader stages are programmable. *Traversal*, which refers to the process of traversing acceleration structures to find potential intersections of rays with geometry, is fixed function.

The programmable portions of the pipeline are exposed in a single-ray programming model. Each GPU thread handles one ray at a time. Memory operations **can** be synchronized using standard memory barriers. However, communication and synchronization between threads is not allowed. In particular, the use of compute pipeline synchronization functions is not supported in the ray tracing pipeline.

To dispatch a ray tracing call use:

```
void vkCmdTraceRaysNV(  
    VkCommandBuffer  
    VkBuffer  
    VkDeviceSize  
    VkBuffer  
    VkDeviceSize  
    VkDeviceSize  
    VkDeviceSize  
    VkBuffer  
    VkDeviceSize  
    VkDeviceSize  
    VkDeviceSize  
    uint32_t  
    uint32_t  
    uint32_t  
        commandBuffer,  
        raygenShaderBindingTableBuffer,  
        raygenShaderBindingOffset,  
        missShaderBindingTableBuffer,  
        missShaderBindingOffset,  
        missShaderBindingStride,  
        hitShaderBindingTableBuffer,  
        hitShaderBindingOffset,  
        hitShaderBindingStride,  
        callableShaderBindingTableBuffer,  
        callableShaderBindingOffset,  
        callableShaderBindingStride,  
        width,  
        height,  
        depth);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **raygenShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the ray generation shader stage.
- **raygenShaderBindingOffset** is the offset in bytes (relative to **raygenShaderBindingTableBuffer**) of the ray generation shader being used for the trace.
- **missShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the miss shader stage.
- **missShaderBindingOffset** is the offset in bytes (relative to **missShaderBindingTableBuffer**) of the miss shader being used for the trace.
- **missShaderBindingStride** is the size in bytes of each shader binding table record in **missShaderBindingTableBuffer**.
- **hitShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the hit shader stages.
- **hitShaderBindingOffset** is the offset in bytes (relative to **hitShaderBindingTableBuffer**) of the hit shader group being used for the trace.
- **hitShaderBindingStride** is the size in bytes of each shader binding table record in **hitShaderBindingTableBuffer**.
- **callableShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the callable shader stage.

- `callableShaderBindingOffset` is the offset in bytes (relative to `callableShaderBindingTableBuffer`) of the callable shader being used for the trace.
- `callableShaderBindingStride` is the size in bytes of each shader binding table record in `callableShaderBindingTableBuffer`.
- `width` is the width of the ray trace query dimensions.
- `height` is height of the ray trace query dimensions.
- `depth` is depth of the ray trace query dimensions.

When the command is executed, a ray generation group of `width` × `height` × `depth` rays is assembled.

## Valid Usage

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN_EXT` or `VK_SAMPLER_REDUCTION_MODE_MAX_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`
- Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.
- For each set  $n$  that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to  $n$  at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set  $n$ , with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in [Pipeline Layout Compatibility](#)
- Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command
- A valid pipeline **must** be bound to the pipeline bind point used by this command
- If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set for `commandBuffer`
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage
- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a

`VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage

- If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point
- If `commandBuffer` is an unprotected command buffer, any resource accessed by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource
- If `commandBuffer` is a protected command buffer, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be an unprotected resource
- If `commandBuffer` is a protected command buffer, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point **must** not write to any resource
- `raygenShaderBindingOffset` **must** be less than the size of `raygenShaderBindingTableBuffer`
- `raygenShaderBindingOffset` **must** be a multiple of `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupBaseAlignment`
- `missShaderBindingOffset` **must** be less than the size of `missShaderBindingTableBuffer`
- `missShaderBindingOffset` **must** be a multiple of `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupBaseAlignment`
- `hitShaderBindingOffset` **must** be less than the size of `hitShaderBindingTableBuffer`
- `hitShaderBindingOffset` **must** be a multiple of `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupBaseAlignment`
- `callableShaderBindingOffset` **must** be less than the size of `callableShaderBindingTableBuffer`
- `callableShaderBindingOffset` **must** be a multiple of `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupBaseAlignment`
- `missShaderBindingStride` **must** be a multiple of `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupHandleSize`
- `hitShaderBindingStride` **must** be a multiple of `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupHandleSize`

- `callableShaderBindingStride` **must** be a multiple of `VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupHandleSize`
- `missShaderBindingStride` **must** be a less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxShaderGroupStride`
- `hitShaderBindingStride` **must** be a less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxShaderGroupStride`
- `callableShaderBindingStride` **must** be a less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxShaderGroupStride`
- `width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`
- `height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]`
- `depth` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `raygenShaderBindingTableBuffer` **must** be a valid `VkBuffer` handle
- If `missShaderBindingTableBuffer` is not `VK_NULL_HANDLE`, `missShaderBindingTableBuffer` **must** be a valid `VkBuffer` handle
- If `hitShaderBindingTableBuffer` is not `VK_NULL_HANDLE`, `hitShaderBindingTableBuffer` **must** be a valid `VkBuffer` handle
- If `callableShaderBindingTableBuffer` is not `VK_NULL_HANDLE`, `callableShaderBindingTableBuffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- This command **must** only be called outside of a render pass instance
- Each of `callableShaderBindingTableBuffer`, `commandBuffer`, `hitShaderBindingTableBuffer`, `missShaderBindingTableBuffer`, and `raygenShaderBindingTableBuffer` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Compute	

## 33.2. Shader Binding Table

A *shader binding table* is a resource which establishes the relationship between the ray tracing pipeline and the acceleration structures that were built for the ray tracing query. It indicates the shaders that operate on each geometry in an acceleration structure. In addition, it contains the resources accessed by each shader, including indices of textures and constants. The application allocates and manages *shader binding tables* as [VkBuffer](#) objects.

Each entry in the shader binding table consists of [shaderGroupHandleSize](#) bytes of data as queried by [vkGetRayTracingShaderGroupHandlesNV](#) to refer to the shader that it invokes. The remainder of the data specified by the stride is application-visible data that can be referenced by a [shaderRecordNV](#) block in the shader.

The shader binding tables to use in a ray tracing query are passed to [vkCmdTraceRaysNV](#). Shader binding tables are read-only in shaders that are executing on the ray tracing pipeline.

### 33.2.1. Indexing Rules

In order to execute the correct shaders and access the correct resources during a ray tracing dispatch, the implementation **must** be able to locate shader binding table entries at various stages of execution. This is accomplished by defining a set of indexing rules that compute shader binding table record positions relative to the buffer's base address in memory. The application **must** organize the contents of the shader binding table's memory in a way that application of the indexing rules will lead to correct records.

#### Ray Generation Shaders

Only one ray generation shader is executed per ray tracing dispatch. Its location is passed into [vkCmdTraceRaysNV](#) using the [raygenShaderBindingTableBuffer](#) and [raygenShaderBindingTableOffset](#) parameters — there is no indexing.

#### Hit Shaders

The base for the computation of intersection, any-hit and closest hit shader locations is the [instanceShaderBindingTableRecordOffset](#) value stored with each instance of a top-level acceleration structure. This value determines the beginning of the shader binding table records for a given instance. Each geometry in the instance **must** have at least one hit program record.

In the following rule, *geometryIndex* refers to the location of the geometry within the instance.

The `sbtRecordStride` and `sbtRecordOffset` values are passed in as parameters to `traceNV()` calls made in the shaders. See Section 8.19 (Ray Tracing Functions) of the OpenGL Shading Language Specification for more details. In SPIR-V, these correspond to the `SBTOffset` and `SBTStride` parameters to the `OpTraceNV` instruction.

The result of this computation is then added to `hitShaderBindingOffset`, a base offset passed to `vkCmdTraceRaysNV`.

The complete rule to compute a hit shader binding table record address in the `hitShaderBindingTableBuffer` is:

$$\text{hitShaderBindingOffset} + \text{hitShaderBindingStride} \times (\text{instanceShaderBindingTableRecordOffset} + \text{geometryIndex} \times \text{sbtRecordStride} + \text{sbtRecordOffset})$$

## Miss Shaders

A miss shader is executed whenever a ray query fails to find an intersection for the given scene geometry. Multiple miss shaders **may** be executed throughout a ray tracing dispatch.

The base for the computation of miss shader locations is `missShaderBindingOffset`, a base offset passed into `vkCmdTraceRaysNV`.

The `missIndex` value is passed in as parameters to `traceNV()` calls made in the shaders. See Section 8.19 (Ray Tracing Functions) of the OpenGL Shading Language Specification for more details. In SPIR-V, this corresponds to the `MissIndex` parameter to the `OpTraceNV` instruction.

The complete rule to compute a miss shader binding table record address in the `missShaderBindingTableBuffer` is:

$$\text{missShaderBindingOffset} + \text{missShaderBindingStride} \times \text{missIndex}$$

## Callable Shaders

A callable shader is executed when requested by a ray tracing shader. Multiple callable shaders **may** be executed throughout a ray tracing dispatch.

The base for the computation of callable shader locations is `callableShaderBindingOffset`, a base offset passed into `vkCmdTraceRaysNV`.

The `sbtRecordIndex` value is passed in as a parameter to `executeCallableNV()` calls made in the shaders. See Section 8.19 (Ray Tracing Functions) of the OpenGL Shading Language Specification for more details. In SPIR-V, this corresponds to the `SBTIndex` parameter to the `OpExecuteCallableNV` instruction.

The complete rule to compute a callable shader binding table record address in the `callableShaderBindingTableBuffer` is:

$$\text{callableShaderBindingOffset} + \text{callableShaderBindingStride} \times \text{sbtRecordIndex}$$

### 33.3. Acceleration Structures

*Acceleration structures* are data structures used by the implementation to efficiently manage the scene geometry as it is traversed during a ray tracing query. The application is responsible for managing acceleration structure objects (see [Acceleration Structures](#), including allocation, destruction, executing builds or updates, and synchronizing resources used during ray tracing queries.

There are two types of acceleration structures, *top level acceleration structures* and *bottom level acceleration structures*.

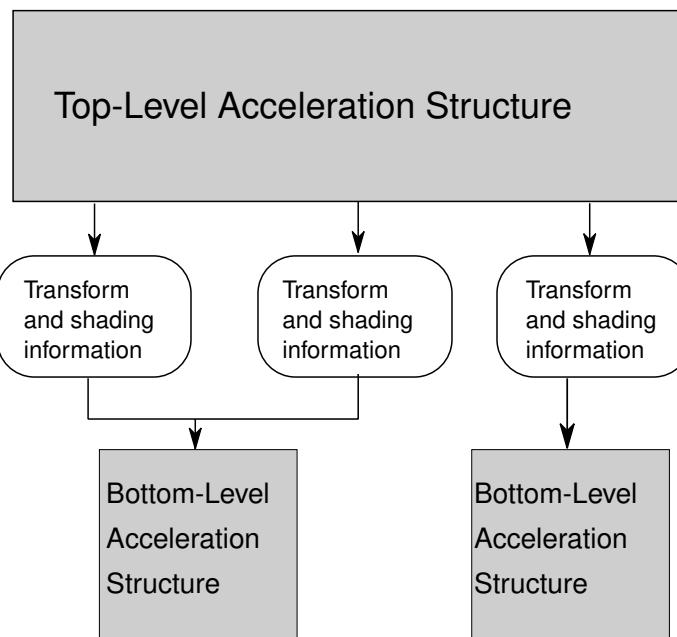


Figure 26. Acceleration Structure

#### Caption

The diagram shows the relationship between top and bottom level acceleration structures.

#### 33.3.1. Instances

*Instances* are found in top level acceleration structures and contain data that refer to a single bottom-level acceleration structure, a transform matrix, and shading information. Multiple instances **can** point to a single bottom level acceleration structure.

An instance is defined in a [VkBuffer](#) by a structure consisting of 64 bytes of data.

- **transform** is 12 floats representing a 4x3 transform matrix in row-major order
- **instanceCustomIndex** The low 24 bits of a 32-bit integer after the transform. This value appears in the builtin `gl_InstanceCustomIndexNV`
- **mask** The high 8 bits of the same integer as `instanceCustomIndex`. This is the visibility mask. The instance **may** only be hit if `rayMask & instance.mask != 0`
- **instanceOffset** The low 24 bits of the next 32-bit integer. The value contributed by this instance

to the hit shader binding table index computation as `instanceShaderBindingTableRecordOffset`.

- `flags` The high 8 bits of the same integer as `instanceOffset`. `VkGeometryInstanceFlagBitsNV` values that apply to this instance.
- `accelerationStructure`. The 8 byte value returned by `vkGetAccelerationStructureHandleNV` for the bottom level acceleration structure referred to by this instance.

*Note*

The C language spec does not define the ordering of bit-fields, but in practice, this struct produces the layout described above:



```
struct VkGeometryInstanceNV {  
    float          transform[12];  
    uint32_t       instanceCustomIndex : 24;  
    uint32_t       mask : 8;  
    uint32_t       instanceOffset : 24;  
    uint32_t       flags : 8;  
    uint64_t       accelerationStructureHandle;  
};
```

Possible values of `flags` in the instance modifying the behavior of that instance are:,

```
typedef enum VkGeometryInstanceFlagBitsNV {  
    VK_GEOMETRY_INSTANCE_TRIANGLE_CULL_DISABLE_BIT_NV = 0x00000001,  
    VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_NV = 0x00000002,  
    VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_NV = 0x00000004,  
    VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_NV = 0x00000008,  
    VK_GEOMETRY_INSTANCE_FLAG_BITS_MAX_ENUM_NV = 0x7FFFFFFF  
} VkGeometryInstanceFlagBitsNV;
```

- `VK_GEOMETRY_INSTANCE_TRIANGLE_CULL_DISABLE_BIT_NV` disables face culling for this instance.
- `VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_NV` indicates that the front face of the triangle for culling purposes is the face that is counter clockwise in object space relative to the ray origin. Because the facing is determined in object space, an instance transform matrix does not change the winding, but a geometry transform does.
- `VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_NV` causes this instance to act as though `VK_GEOMETRY_OPAQUE_BIT_NV` were specified on all geometries referenced by this instance. This behavior **can** be overridden by the ray flag `gl_RayFlagsNoOpaqueNV`.
- `VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_NV` causes this instance to act as though `VK_GEOMETRY_OPAQUE_BIT_NV` were not specified on all geometries referenced by this instance. This behavior **can** be overridden by the ray flag `gl_RayFlagsOpaqueNV`.

`VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_NV` and `VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_NV` **must** not be used in the same flag.

```
typedef VkFlags VkGeometryInstanceFlagsNV;
```

`VkGeometryInstanceFlagsNV` is a bitmask type for setting a mask of zero or more `VkGeometryInstanceFlagBitsNV`.

### 33.3.2. Geometry

*Geometries* refer to a triangle or axis-aligned bounding box.

### 33.3.3. Top Level Acceleration Structures

Opaque acceleration structure for an array of instances. The descriptor referencing this is the starting point for tracing

### 33.3.4. Bottom Level Acceleration Structures

Opaque acceleration structure for an array of geometries.

### 33.3.5. Building Acceleration Structures

To build an acceleration structure call:

```
void vkCmdBuildAccelerationStructureNV(  
    VkCommandBuffer  
    const VkAccelerationStructureCreateInfoNV*  
    VkBuffer  
    VkDeviceSize  
    VkBool32  
    VkAccelerationStructureNV  
    VkAccelerationStructureNV  
    VkBuffer  
    VkDeviceSize  
    commandBuffer,  
    pInfo,  
    instanceData,  
    instanceOffset,  
    update,  
    dst,  
    src,  
    scratch,  
    scratchOffset);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pInfo` contains the shared information for the acceleration structure's structure.
- `instanceData` is the buffer containing instance data that will be used to build the acceleration structure as described in [Accelerator structure instances](#). This parameter **must** be `NULL` for bottom level acceleration structures.
- `instanceOffset` is the offset in bytes (relative to the start of `instanceData`) at which the instance data is located.
- `update` specifies whether to update the `dst` acceleration structure with the data in `src`.
- `dst` is a pointer to the target acceleration structure for the build.
- `src` is a pointer to an existing acceleration structure that is to be used to update the `dst` acceleration structure.

- `scratch` is the `VkBuffer` that will be used as scratch memory for the build.
- `scratchOffset` is the offset in bytes relative to the start of `scratch` that will be used as a scratch memory.

## Valid Usage

- `geometryCount` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxGeometryCount`
- `dst` **must** have been created with compatible `VkAccelerationStructureInfoNV` where `VkAccelerationStructureInfoNV::type` and `VkAccelerationStructureInfoNV::flags` are identical, `VkAccelerationStructureInfoNV::instanceCount` and `VkAccelerationStructureInfoNV::geometryCount` for `dst` are greater than or equal to the build size and each geometry in `VkAccelerationStructureInfoNV::pGeometries` for `dst` has greater than or equal to the number of vertices, indices, and AABBs.
- If `update` is `VK_TRUE`, `src` **must** not be `VK_NULL_HANDLE`
- If `update` is `VK_TRUE`, `src` **must** have been built before with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV` set in `VkAccelerationStructureInfoNV::flags`
- If `update` is `VK_FALSE`, The `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetAccelerationStructureMemoryRequirementsNV` with `VkAccelerationStructureMemoryRequirementsInfoNV::accelerationStructure` set to `dst` and `VkAccelerationStructureMemoryRequirementsInfoNV::type` set to `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_BUILD_SCRATCH_NV` **must** be less than or equal to the size of `scratch` minus `scratchOffset`
- If `update` is `VK_TRUE`, The `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetAccelerationStructureMemoryRequirementsNV` with `VkAccelerationStructureMemoryRequirementsInfoNV::accelerationStructure` set to `dst` and `VkAccelerationStructureMemoryRequirementsInfoNV::type` set to `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_UPDATE_SCRATCH_NV` **must** be less than or equal to the size of `scratch` minus `scratchOffset`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pInfo` **must** be a valid pointer to a valid `VkAccelerationStructureCreateInfoNV` structure
- If `instanceData` is not `VK_NULL_HANDLE`, `instanceData` **must** be a valid `VkBuffer` handle
- `dst` **must** be a valid `VkAccelerationStructureNV` handle
- If `src` is not `VK_NULL_HANDLE`, `src` **must** be a valid `VkAccelerationStructureNV` handle
- `scratch` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- This command **must** only be called outside of a render pass instance
- Each of `commandBuffer`, `dst`, `instanceData`, `scratch`, and `src` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Compute	

### 33.3.6. Copying Acceleration Structures

An additional command exists for copying acceleration structures without updating their contents. The acceleration structure object **can** be compacted in order to improve performance. Before copying, an application **must** query the size of the resulting acceleration structure.

To query acceleration structure size parameters call:

```
void vkCmdWriteAccelerationStructuresPropertiesNV(  
    VkCommandBuffer commandBuffer,  
    uint32_t accelerationStructureCount,  
    const VkAccelerationStructureNV* pAccelerationStructures,  
    VkQueryType queryType,  
    VkQueryPool queryPool,  
    uint32_t firstQuery);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `accelerationStructureCount` is the count of acceleration structures for which to query the property.
- `pAccelerationStructures` is a pointer to an array of existing previously built acceleration structures.
- `queryType` is a `VkQueryType` value specifying the type of queries managed by the pool.
- `queryPool` is the query pool that will manage the results of the query.
- `firstQuery` is the first query index within the query pool that will contain the `accelerationStructureCount` number of results.

## Valid Usage

- `queryType` **must** be `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV`
- `queryPool` **must** have been created with a `queryType` matching `queryType`
- The queries identified by `queryPool` and `firstQuery` **must** be *Unavailable*
- All acceleration structures in `accelerationStructures` **must** have been built with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_NV` if `queryType` is `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pAccelerationStructures` **must** be a valid pointer to an array of `accelerationStructureCount` valid `VkAccelerationStructureNV` handles
- `queryType` **must** be a valid `VkQueryType` value
- `queryPool` **must** be a valid `VkQueryPool` handle
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- This command **must** only be called outside of a render pass instance
- `accelerationStructureCount` **must** be greater than `0`
- Each of `commandBuffer`, `queryPool`, and the elements of `pAccelerationStructures` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary	Inside	Graphics	Graphics Pipeline
Secondary	Outside	Compute	Compute Pipeline

To copy an acceleration structure call:

```
void vkCmdCopyAccelerationStructureNV(  
    VkCommandBuffer  
    VkAccelerationStructureNV  
    VkAccelerationStructureNV  
    VkCopyAccelerationStructureModeNV  
        commandBuffer,  
        dst,  
        src,  
        mode);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `dst` is a pointer to the target acceleration structure for the copy.
- `src` is a pointer to the source acceleration structure for the copy.
- `mode` is a `VkCopyAccelerationStructureModeNV` value specifying additional operations to

perform during the copy.

## Valid Usage

- `mode` **must** be `VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_NV` or `VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_NV`
- `src` **must** have been built with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_NV` if `mode` is `VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_NV`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `dst` **must** be a valid `VkAccelerationStructureNV` handle
- `src` **must** be a valid `VkAccelerationStructureNV` handle
- `mode` **must** be a valid `VkCopyAccelerationStructureModeNV` value
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations
- This command **must** only be called outside of a render pass instance
- Each of `commandBuffer`, `dst`, and `src` **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Outside	Compute	

Possible values of `vkCmdCopyAccelerationStructureNV::mode`, specifying additional operations to perform during the copy, are:

```
typedef enum VkCopyAccelerationStructureModeNV {
    VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_NV = 0,
    VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_NV = 1,
    VK_COPY_ACCELERATION_STRUCTURE_MODE_MAX_ENUM_NV = 0x7FFFFFFF
} VkCopyAccelerationStructureModeNV;
```

- `VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_NV` creates a direct copy of the acceleration structure specified in `src` into the one specified by `dst`. The `dst` acceleration structure **must** have been created with the same parameters as `src`.
- `VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_NV` creates a more compact version of an acceleration structure `src` into `dst`. The acceleration structure `dst` **must** have been created with a `compactSize` corresponding to the one returned by `vkCmdWriteAccelerationStructuresPropertiesNV` after the build of the acceleration structure specified by `src`.

# Chapter 34. Extending Vulkan

New functionality **may** be added to Vulkan via either new extensions or new versions of the core, or new versions of an extension in some cases.

This chapter describes how Vulkan is versioned, how compatibility is affected between different versions, and compatibility rules that are followed by the Vulkan Working Group.

## 34.1. Instance and Device Functionality

Commands that enumerate instance properties, or that accept a `VkInstance` object as a parameter, are considered instance-level functionality. Commands that enumerate physical device properties, or that accept a `VkDevice` object or any of a device's child objects as a parameter, are considered device-level functionality.

*Note*

Applications usually interface to Vulkan using a loader that implements only instance-level functionality, passing device-level functionality to implementations of the full Vulkan API on the system. In some circumstances, as these may be implemented independently, it is possible that the loader and device implementations on a given installation will support different versions. To allow for this and call out when it happens, the Vulkan specification enumerates device and instance level functionality separately - they have [independent version queries](#).



*Note*

Vulkan 1.0 initially specified new physical device enumeration functionality as instance-level, requiring it to be included in an instance extension. As the capabilities of device-level functionality require discovery via physical device enumeration, this led to the situation where many device extensions required an instance extension as well. To alleviate this extra work, [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#) (and subsequently Vulkan 1.1) redefined device-level functionality to include physical device enumeration.



## 34.2. Core Versions

The Vulkan Specification is regularly updated with bug fixes and clarifications. Occasionally new functionality is added to the core and at some point it is expected that there will be a desire to perform a large, breaking change to the API. In order to indicate to developers how and when these changes are made to the specification, and to provide a way to identify each set of changes, the Vulkan API maintains a version number.

### 34.2.1. Version Numbers

The Vulkan version number comprises three parts indicating the major, minor and patch version of the Vulkan API Specification.

The *major version* indicates a significant change in the API, which will encompass a wholly new version of the specification.

The *minor version* indicates the incorporation of new functionality into the core specification.

The *patch version* indicates bug fixes, clarifications, and language improvements have been incorporated into the specification.

Compatibility guarantees made about versions of the API sharing any of the same version numbers are documented in [Core Versions](#)

The version number is used in several places in the API. In each such use, the version numbers are packed into a 32-bit integer as follows:

- The major version is a 10-bit integer packed into bits 31-22.
- The minor version number is a 10-bit integer packed into bits 21-12.
- The patch version number is a 12-bit integer packed into bits 11-0.

**VK\_VERSION\_MAJOR** extracts the API major version number from a packed version number:

```
#define VK_VERSION_MAJOR(version) ((uint32_t)(version) >> 22)
```

**VK\_VERSION\_MINOR** extracts the API minor version number from a packed version number:

```
#define VK_VERSION_MINOR(version) (((uint32_t)(version) >> 12) & 0x3ff)
```

**VK\_VERSION\_PATCH** extracts the API patch version number from a packed version number:

```
#define VK_VERSION_PATCH(version) ((uint32_t)(version) & 0xffff)
```

**VK\_MAKE\_VERSION** constructs an API version number.

```
#define VK_MAKE_VERSION(major, minor, patch) \  
    (((major) << 22) | ((minor) << 12) | (patch))
```

- **major** is the major version number.
- **minor** is the minor version number.
- **patch** is the patch version number.

**VK\_API\_VERSION\_1\_0** returns the API version number for Vulkan 1.0.0.

```
// Vulkan 1.0 version number
#define VK_API_VERSION_1_0 VK_MAKE_VERSION(1, 0, 0)// Patch version should always be
set to 0
```

`VK_API_VERSION_1_1` returns the API version number for Vulkan 1.1.0.

```
// Vulkan 1.1 version number
#define VK_API_VERSION_1_1 VK_MAKE_VERSION(1, 1, 0)// Patch version should always be
set to 0
```

### 34.2.2. Querying Version Support

The version of instance-level functionality can be queried by calling `vkEnumerateInstanceVersion`.

The version of device-level functionality can be queried by calling `vkGetPhysicalDeviceProperties` or `vkGetPhysicalDeviceProperties2`, and is returned in `VkPhysicalDeviceProperties::apiVersion`, encoded as described in [Version Numbers](#).

## 34.3. Layers

When a layer is enabled, it inserts itself into the call chain for Vulkan commands the layer is interested in. Layers **can** be used for a variety of tasks that extend the base behavior of Vulkan beyond what is required by the specification - such as call logging, tracing, validation, or providing additional extensions.

*Note*

For example, an implementation is not expected to check that the value of enums used by the application fall within allowed ranges. Instead, a validation layer would do those checks and flag issues. This avoids a performance penalty during production use of the application because those layers would not be enabled in production.

*Note*

Vulkan layers **may** wrap object handles (i.e. return a different handle value to the application than that generated by the implementation). This is generally discouraged, as it increases the probability of incompatibilities with new extensions. The validation layers wrap handles in order to track the proper use and destruction of each object. See the [“Vulkan Loader Specification and Architecture Overview”](#) document for additional information.

To query the available layers, call:

```
VkResult vkEnumerateInstanceLayerProperties(  
    uint32_t*  
    VkLayerProperties*  
        pPropertyCount,  
        pProperties);
```

- `pPropertyCount` is a pointer to an integer related to the number of layer properties available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkLayerProperties` structures.

If `pProperties` is `NULL`, then the number of layer properties available is returned in `pPropertyCount`. Otherwise, `pPropertyCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If `pPropertyCount` is less than the number of layer properties available, at most `pPropertyCount` structures will be written. If `pPropertyCount` is smaller than the number of layers available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available layer properties were returned.

The list of available layers may change at any time due to actions outside of the Vulkan implementation, so two calls to `vkEnumerateInstanceLayerProperties` with the same parameters **may** return different results, or retrieve different `pPropertyCount` values or `pProperties` contents. Once an instance has been created, the layers enabled for that instance will continue to be enabled and valid for the lifetime of that instance, even if some of them become unavailable for future instances.

## Valid Usage (Implicit)

- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkLayerProperties` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkLayerProperties` structure is defined as:

```
typedef struct VkLayerProperties {
    char      layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t   specVersion;
    uint32_t   implementationVersion;
    char      description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

- `layerName` is an array of `VK_MAX_EXTENSION_NAME_SIZE` `char` containing a null-terminated UTF-8 string which is the name of the layer. Use this name in the `ppEnabledLayerNames` array passed in the `VkInstanceCreateInfo` structure to enable this layer for an instance.
- `specVersion` is the Vulkan version the layer was written to, encoded as described in [Version Numbers](#).
- `implementationVersion` is the version of this layer. It is an integer, increasing with backward compatible changes.
- `description` is an array of `VK_MAX_DESCRIPTION_SIZE` `char` containing a null-terminated UTF-8 string which provides additional details that **can** be used by the application to identify the layer.

To enable a layer, the name of the layer **should** be added to the `ppEnabledLayerNames` member of `VkInstanceCreateInfo` when creating a `VkInstance`.

Loader implementations **may** provide mechanisms outside the Vulkan API for enabling specific layers. Layers enabled through such a mechanism are *implicitly enabled*, while layers enabled by including the layer name in the `ppEnabledLayerNames` member of `VkInstanceCreateInfo` are *explicitly enabled*. Except where otherwise specified, implicitly enabled and explicitly enabled layers differ only in the way they are enabled. Explicitly enabling a layer that is implicitly enabled has no additional effect.

### 34.3.1. Device Layer Deprecation

Previous versions of this specification distinguished between instance and device layers. Instance layers were only able to intercept commands that operate on `VkInstance` and `VkPhysicalDevice`, except they were not able to intercept `vkCreateDevice`. Device layers were enabled for individual devices when they were created, and could only intercept commands operating on that device or its child objects.

Device-only layers are now deprecated, and this specification no longer distinguishes between instance and device layers. Layers are enabled during instance creation, and are able to intercept all commands operating on that instance or any of its child objects. At the time of deprecation there were no known device-only layers and no compelling reason to create one.

In order to maintain compatibility with implementations released prior to device-layer deprecation, applications **should** still enumerate and enable device layers. The behavior of `vkEnumerateDeviceLayerProperties` and valid usage of the `ppEnabledLayerNames` member of `VkDeviceCreateInfo` maximizes compatibility with applications written to work with the previous requirements.

To enumerate device layers, call:

```
VkResult vkEnumerateDeviceLayerProperties(  
    VkPhysicalDevice physicalDevice,  
    uint32_t* pPropertyCount,  
    VkLayerProperties* pProperties);
```

- `pPropertyCount` is a pointer to an integer related to the number of layer properties available or queried.
- `pProperties` is either `NULL` or a pointer to an array of `VkLayerProperties` structures.

If `pProperties` is `NULL`, then the number of layer properties available is returned in `pPropertyCount`. Otherwise, `pPropertyCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If `pPropertyCount` is less than the number of layer properties available, at most `pPropertyCount` structures will be written. If `pPropertyCount` is smaller than the number of layers available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available layer properties were returned.

The list of layers enumerated by `vkEnumerateDeviceLayerProperties` **must** be exactly the sequence of layers enabled for the instance. The members of `VkLayerProperties` for each enumerated layer **must** be the same as the properties when the layer was enumerated by `vkEnumerateInstanceLayerProperties`.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkLayerProperties` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `ppEnabledLayerNames` and `enabledLayerCount` members of `VkDeviceCreateInfo` are deprecated and their values **must** be ignored by implementations. However, for compatibility, only an empty list of layers or a list that exactly matches the sequence enabled at instance creation time are valid, and validation layers **should** issue diagnostics for other cases.

Regardless of the enabled layer list provided in `VkDeviceCreateInfo`, the sequence of layers active for a device will be exactly the sequence of layers enabled when the parent instance was created.

## 34.4. Extensions

Extensions **may** define new Vulkan commands, structures, and enumerants. For compilation purposes, the interfaces defined by registered extensions, including new structures and enumerants as well as function pointer types for new commands, are defined in the Khronos-supplied `vulkan_core.h` together with the core API. However, commands defined by extensions **may** not be available for static linking - in which case function pointers to these commands **should** be queried at runtime as described in [Command Function Pointers](#). Extensions **may** be provided by layers as well as by a Vulkan implementation.

Because extensions **may** extend or change the behavior of the Vulkan API, extension authors **should** add support for their extensions to the Khronos validation layers. This is especially important for new commands whose parameters have been wrapped by the validation layers. See the “[Vulkan Loader Specification and Architecture Overview](#)” document for additional information.

*Note*



Valid Usage sections for individual commands and structures do not currently contain which extensions have to be enabled in order to make their use valid, although it might do so in the future. It is defined only in the [Valid Usage for Extensions](#) section.

### 34.4.1. Instance Extensions

Instance extensions add new [instance-level functionality](#) to the API, outside of the core specification.

To query the available instance extensions, call:

```
VkResult vkEnumerateInstanceExtensionProperties(  
    const char* pLayerName,  
    uint32_t* pPropertyCount,  
    VkExtensionProperties* pProperties);
```

- `pLayerName` is either `NULL` or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- `pPropertyCount` is a pointer to an integer related to the number of extension properties available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkExtensionProperties` structures.

When `pLayerName` parameter is `NULL`, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When `pLayerName` is the name of a layer, the instance extensions provided by that layer are returned.

If `pProperties` is `NULL`, then the number of extensions properties available is returned in

`pPropertyCount`. Otherwise, `pPropertyCount` **must** point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If `pPropertyCount` is less than the number of extension properties available, at most `pPropertyCount` structures will be written. If `pPropertyCount` is smaller than the number of extensions available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available properties were returned.

Because the list of available layers may change externally between calls to `vkEnumerateInstanceExtensionProperties`, two calls may retrieve different results if a `pLayerName` is available in one call but not in another. The extensions supported by a layer may also change between two calls, e.g. if the layer implementation is replaced by a different version between those calls.

Implementations **must** not advertise any pair of extensions that cannot be enabled together due to behavioral differences, or any extension that cannot be enabled against the advertised version.

## Valid Usage (Implicit)

- If `pLayerName` is not `NULL`, `pLayerName` **must** be a null-terminated UTF-8 string
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkExtensionProperties` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_LAYER_NOT_PRESENT`

To enable an instance extension, the name of the extension **should** be added to the `ppEnabledExtensionNames` member of `VkInstanceCreateInfo` when creating a `VkInstance`.

#### Note



Enabling an extension does not change behavior of functionality exposed by the core Vulkan API or any other extension, other than making valid the use of the commands, enums and structures defined by that extension.

### 34.4.2. Device Extensions

Device extensions add new [device-level functionality](#) to the API, outside of the core specification.

To query the extensions available to a given physical device, call:

```
VkResult vkEnumerateDeviceExtensionProperties(  
    VkPhysicalDevice physicalDevice,  
    const char* pLayerName,  
    uint32_t* pPropertyCount,  
    VkExtensionProperties* pProperties);
```

- `physicalDevice` is the physical device that will be queried.
- `pLayerName` is either `NULL` or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- `pPropertyCount` is a pointer to an integer related to the number of extension properties available or queried, and is treated in the same fashion as the `vkEnumerateInstanceExtensionProperties` `::pPropertyCount` parameter.
- `pProperties` is either `NULL` or a pointer to an array of `VkExtensionProperties` structures.

When `pLayerName` parameter is `NULL`, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When `pLayerName` is the name of a layer, the device extensions provided by that layer are returned.

Implementations **must** not advertise any pair of extensions that cannot be enabled together due to behavioral differences, or any extension that cannot be enabled against the advertised version.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- If `pLayerName` is not `NULL`, `pLayerName` **must** be a null-terminated UTF-8 string
- `pPropertyCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkExtensionProperties` structures

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_LAYER_NOT_PRESENT`

The `VkExtensionProperties` structure is defined as:

```
typedef struct VkExtensionProperties {
    char      extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t   specVersion;
} VkExtensionProperties;
```

- `extensionName` is an array of `VK_MAX_EXTENSION_NAME_SIZE` `char` containing a null-terminated UTF-8 string which is the name of the extension.
- `specVersion` is the version of this extension. It is an integer, incremented with backward compatible changes.

## 34.5. Extension Dependencies

Some extensions are dependent on other extensions to function. To enable extensions with dependencies, such *required extensions* **must** also be enabled through the same API mechanisms when creating an instance with `vkCreateInstance` or a device with `vkCreateDevice`. Each extension which has such dependencies documents them in the [appendix summarizing that extension](#).

If an extension is supported (as queried by `vkEnumerateInstanceExtensionProperties` or `vkEnumerateDeviceExtensionProperties`), then *required extensions* of that extension **must** also be supported for the same instance or physical device.

Any device extension that has an instance extension dependency that is not enabled by `vkCreateInstance` is considered to be unsupported, hence it **must** not be returned by `vkEnumerateDeviceExtensionProperties` for any `VkPhysicalDevice` child of the instance.

## 34.6. Compatibility Guarantees (Informative)

This section is marked as informal as there is no binding responsibility on implementations of the Vulkan API - these guarantees are however a contract between the Vulkan Working Group and developers using this Specification.

### 34.6.1. Core Versions

Each of the [major, minor, and patch versions](#) of the Vulkan specification provide different compatibility guarantees.

#### Patch Versions

A difference in the patch version indicates that a set of bug fixes or clarifications have been made to the Specification. Informative enums returned by Vulkan commands that will not affect the runtime behavior of a valid application may be added in a patch version (e.g. `VkVendorId`).

The specification's patch version is strictly increasing for a given major version of the specification; any change to a specification as described above will result in the patch version being increased by 1. Patch versions are applied to all minor versions, even if a given minor version is not affected by the provoking change.

Specifications with different patch versions but the same major and minor version are *fully compatible* with each other - such that a valid application written against one will work with an implementation of another.

#### Note



If a patch version includes a bug fix or clarification that could have a significant impact on developer expectations, these will be highlighted in the change log. Generally the Vulkan Working Group tries to avoid these kinds of changes, instead fixing them in either an extension or core version.

## Minor Versions

Changes in the minor version of the specification indicate that new functionality has been added to the core specification. This will usually include new interfaces in the header, and **may** also include behavior changes and bug fixes. Core functionality **may** be deprecated in a minor version, but will not be obsoleted or removed.

The specification's minor version is strictly increasing for a given major version of the specification; any change to a specification as described above will result in the minor version being increased by 1. Changes that can be accommodated in a patch version will not increase the minor version.

Specifications with a lower minor version are *backwards compatible* with an implementation of a specification with a higher minor version for core functionality and extensions issued with the KHR vendor tag. Vendor and multi-vendor extensions are not guaranteed to remain functional across minor versions, though in general they are with few exceptions - see [Obsoletion](#) for more information.

## Major Versions

A difference in the major version of specifications indicates a large set of changes which will likely include interface changes, behavioral changes, removal of [deprecated functionality](#), and the modification, addition, or replacement of other functionality.

The specification's major version is monotonically increasing; any change to the specification as described above will result in the major version being increased. Changes that can be accommodated in a patch or minor version will not increase the major version.

The Vulkan Working Group intends to only issue a new major version of the Specification in order to realise significant improvements to the Vulkan API that will necessarily require breaking compatibility.

A new major version will likely include a wholly new version of the specification to be issued - which could include an overhaul of the versioning semantics for the minor and patch versions. The patch and minor versions of a specification are therefore not meaningful across major versions. If a major version of the specification includes similar versioning semantics, it is expected that the patch and minor version will be reset to 0 for that major version.

## 34.6.2. Extensions

A KHR extension **must** be able to be enabled alongside any other KHR extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A multi-vendor extension **should** be able to be enabled alongside any KHR extension or other multi-vendor extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension **should** be able to be enabled alongside any KHR extension, multi-vendor extension, or other vendor extension from the same vendor, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension **may** be able to be enabled alongside vendor extensions from another vendor.

The one other exception to this is if a vendor or multi-vendor extension is [made obsolete](#) by either a core version or another extension, which will be highlighted in the [extension appendix](#).

### Promotion

Extensions, or features of an extension, **may** be promoted to a new [core version of the API](#), or a newer extension which an equal or greater number of implementors are in favour of.

When extension functionality is promoted, minor changes **may** be introduced, limited to the following:

- Naming
- Non-intrusive parameters changes
- [Feature advertisement/enablement](#)
- Combining structure parameters into larger structures
- Author ID suffixes changed or removed

#### *Note*

If extension functionality is promoted, there is no guarantee of direct compatibility, however it should require little effort to port code from the original feature to the promoted one.



The Vulkan Working Group endeavours to ensure that larger changes are marked as either [deprecated](#) or [obsoleted](#) as appropriate, and can do so retroactively if necessary.

Extensions that are promoted are listed as being promoted in their extension appendices, with reference to where they were promoted to.

When an extension is promoted, any backwards compatibility aliases which exist in the extension will **not** be promoted.

### *Note*

As a hypothetical example, if the `VK_KHR_surface` extension were promoted to part of a future core version, the `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR` token defined by that extension would be promoted to `VK_COLOR_SPACE_SRGB_NONLINEAR`. However, the `VK_COLORSPACE_SRGB_NONLINEAR_KHR` token aliases `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`. The `VK_COLORSPACE_SRGB_NONLINEAR_KHR` would not be promoted, because it is a backwards compatibility alias that exists only due to a naming mistake when the extension was initially published.



## Deprecation

Extensions **may** be marked as deprecated when the intended use cases either become irrelevant or can be solved in other ways. Generally, a new feature will become available to solve the use case in another extension or core version of the API, but it is not guaranteed.

### *Note*



Features that are intended to replace deprecated functionality have no guarantees of compatibility, and applications may require drastic modification in order to make use of the new features.

Extensions that are deprecated are listed as being deprecated in their extension appendices, with an explanation of the deprecation and any features that are relevant.

## Obsoletion

Occasionally, an extension will be marked as obsolete if a new version of the core API or a new extension is fundamentally incompatible with it. An obsoleted extension **must** not be used with the extension or core version that obsoleted it.

Extensions that are obsoleted are listed as being obsoleted in their extension appendices, with reference to what they were obsoleted by.

## Aliases

When an extension is promoted or deprecated by a newer feature, some or all of its functionality **may** be replicated into the newer feature. Rather than duplication of all the documentation and definitions, the specification instead identifies the identical commands and types as *aliases* of one another. Each alias is mentioned together with the definition it aliases, with the older aliases marked as “equivalents”. Each alias of the same command has identical behavior, and each alias of the same type has identical meaning - they can be used interchangably in an application with no compatibility issues.

*Note*

For promoted types, the aliased extension type is semantically identical to the new core type. The C99 headers simply **typedef** the older aliases to the promoted types.



For promoted command aliases, however, there are two separate entry point definitions, due to the fact that the C99 ABI has no way to alias command definitions without resorting to macros. Calling via either entry point definition will produce identical behavior within the bounds of the specification, and should still invoke the same entry point in the implementation. Debug tools may use separate entry points with different debug behavior; to write the appropriate command name to an output log, for instance.

# Chapter 35. Features

*Features* describe functionality which is not supported on all implementations. Features are properties of the physical device. Features are **optional**, and **must** be explicitly enabled before use. Support for features is reported and enabled on a per-feature basis.

## Note



Features are reported via the basic `VkPhysicalDeviceFeatures` structure, as well as the extensible structure `VkPhysicalDeviceFeatures2`, which was added in the `VK_KHR_get_physical_device_properties2` extension and included in Vulkan 1.1. When new features are added in future Vulkan versions or extensions, each extension **should** introduce one new feature structure, if needed. This structure **can** be added to the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure.

To query supported features, call:

```
void vkGetPhysicalDeviceFeatures(  
    VkPhysicalDevice           physicalDevice,  
    VkPhysicalDeviceFeatures*  pFeatures);
```

- `physicalDevice` is the physical device from which to query the supported features.
- `pFeatures` is a pointer to a `VkPhysicalDeviceFeatures` structure in which the physical device features are returned. For each feature, a value of `VK_TRUE` specifies that the feature is supported on this physical device, and `VK_FALSE` specifies that the feature is not supported.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pFeatures` **must** be a valid pointer to a `VkPhysicalDeviceFeatures` structure

Fine-grained features used by a logical device **must** be enabled at `VkDevice` creation time. If a feature is enabled that the physical device does not support, `VkDevice` creation will fail and return `VK_ERROR_FEATURE_NOT_PRESENT`.

The fine-grained features are enabled by passing a pointer to the `VkPhysicalDeviceFeatures` structure via the `pEnabledFeatures` member of the `VkDeviceCreateInfo` structure that is passed into the `vkCreateDevice` call. If a member of `pEnabledFeatures` is set to `VK_TRUE` or `VK_FALSE`, then the device will be created with the indicated feature enabled or disabled, respectively. Features **can** also be enabled by using the `VkPhysicalDeviceFeatures2` structure.

If an application wishes to enable all features supported by a device, it **can** simply pass in the `VkPhysicalDeviceFeatures` structure that was previously returned by `vkGetPhysicalDeviceFeatures`. To disable an individual feature, the application **can** set the desired member to `VK_FALSE` in the same structure. Setting `pEnabledFeatures` to `NULL` and not including a `VkPhysicalDeviceFeatures2` in the `pNext` member of `VkDeviceCreateInfo` is equivalent to setting all members of the structure to

`VK_FALSE`.

*Note*



Some features, such as `robustBufferAccess`, **may** incur a run-time performance cost. Application writers **should** carefully consider the implications of enabling all supported features.

To query supported features defined by the core or extensions, call:

```
void vkGetPhysicalDeviceFeatures2(  
    VkPhysicalDevice           physicalDevice,  
    VkPhysicalDeviceFeatures2* pFeatures);
```

or the equivalent command

```
void vkGetPhysicalDeviceFeatures2KHR(  
    VkPhysicalDevice           physicalDevice,  
    VkPhysicalDeviceFeatures2* pFeatures);
```

- `physicalDevice` is the physical device from which to query the supported features.
- `pFeatures` is a pointer to a `VkPhysicalDeviceFeatures2` structure in which the physical device features are returned.

Each structure in `pFeatures` and its `pNext` chain contain members corresponding to fine-grained features. `vkGetPhysicalDeviceFeatures2` writes each member to a boolean value indicating whether that feature is supported.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pFeatures` **must** be a valid pointer to a `VkPhysicalDeviceFeatures2` structure

The `VkPhysicalDeviceFeatures2` structure is defined as:

```
typedef struct VkPhysicalDeviceFeatures2 {  
    VkStructureType      sType;  
    void*               pNext;  
    VkPhysicalDeviceFeatures features;  
} VkPhysicalDeviceFeatures2;
```

or the equivalent

```
typedef VkPhysicalDeviceFeatures2 VkPhysicalDeviceFeatures2KHR;
```

The `VkPhysicalDeviceFeatures2` structure is defined as:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `features` is a `VkPhysicalDeviceFeatures` structure describing the fine-grained features of the Vulkan 1.0 API.

The `pNext` chain of this structure is used to extend the structure with features defined by extensions. This structure **can** be used in `vkGetPhysicalDeviceFeatures2` or **can** be in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which features are enabled in the device in lieu of `pEnabledFeatures`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2`

The `VkPhysicalDeviceFeatures` structure is defined as:

```
typedef struct VkPhysicalDeviceFeatures {  
    VkBool32    robustBufferAccess;  
    VkBool32    fullDrawIndexUint32;  
    VkBool32    imageCubeArray;  
    VkBool32    independentBlend;  
    VkBool32    geometryShader;  
    VkBool32    tessellationShader;  
    VkBool32    sampleRateShading;  
    VkBool32    dualSrcBlend;  
    VkBool32    logicOp;  
    VkBool32    multiDrawIndirect;  
    VkBool32    drawIndirectFirstInstance;  
    VkBool32    depthClamp;  
    VkBool32    depthBiasClamp;  
    VkBool32    fillModeNonSolid;  
    VkBool32    depthBounds;  
    VkBool32    wideLines;  
    VkBool32    largePoints;  
    VkBool32    alphaToOne;  
    VkBool32    multiViewport;  
    VkBool32    samplerAnisotropy;  
    VkBool32    textureCompressionETC2;  
    VkBool32    textureCompressionASTC_LDR;  
    VkBool32    textureCompressionBC;  
    VkBool32    occlusionQueryPrecise;  
    VkBool32    pipelineStatisticsQuery;  
    VkBool32    vertexPipelineStoresAndAtomics;  
    VkBool32    fragmentStoresAndAtomics;  
    VkBool32    shaderTessellationAndGeometryPointSize;  
    VkBool32    shaderImageGatherExtended;
```

```

VkBool32    shaderStorageImageExtendedFormats;
VkBool32    shaderStorageImageMultisample;
VkBool32    shaderStorageImageReadWithoutFormat;
VkBool32    shaderStorageImageWriteWithoutFormat;
VkBool32    shaderUniformBufferArrayDynamicIndexing;
VkBool32    shaderSampledImageArrayDynamicIndexing;
VkBool32    shaderStorageBufferArrayDynamicIndexing;
VkBool32    shaderStorageImageArrayDynamicIndexing;
VkBool32    shaderClipDistance;
VkBool32    shaderCullDistance;
VkBool32    shaderFloat64;
VkBool32    shaderInt64;
VkBool32    shaderInt16;
VkBool32    shaderResourceResidency;
VkBool32    shaderResourceMinLod;
VkBool32    sparseBinding;
VkBool32    sparseResidencyBuffer;
VkBool32    sparseResidencyImage2D;
VkBool32    sparseResidencyImage3D;
VkBool32    sparseResidency2Samples;
VkBool32    sparseResidency4Samples;
VkBool32    sparseResidency8Samples;
VkBool32    sparseResidency16Samples;
VkBool32    sparseResidencyAliased;
VkBool32    variableMultisampleRate;
VkBool32    inheritedQueries;
} VkPhysicalDeviceFeatures;

```

The members of the `VkPhysicalDeviceFeatures` structure describe the following features:

- `robustBufferAccess` specifies that accesses to buffers are bounds-checked against the range of the buffer descriptor (as determined by `VkDescriptorBufferInfo::range`, `VkBufferViewCreateInfo ::range`, or the size of the buffer). Out of bounds accesses **must** not cause application termination, and the effects of shader loads, stores, and atomics **must** conform to an implementation-dependent behavior as described below.
  - A buffer access is considered to be out of bounds if any of the following are true:
    - The pointer was formed by `OpImageTexelPointer` and the coordinate is less than zero or greater than or equal to the number of whole elements in the bound range.
    - The pointer was not formed by `OpImageTexelPointer` and the object pointed to is not wholly contained within the bound range. This includes accesses performed via *variable pointers* where the buffer descriptor being accessed cannot be statically determined. Uninitialized pointers and pointers equal to `OpConstantNull` are treated as pointing to a zero-sized object, so all accesses through such pointers are considered to be out of bounds. Buffer accesses through buffer device addresses are not bounds-checked. If the `cooperativeMatrixRobustBufferAccess` feature is not enabled, then accesses using `OpCooperativeMatrixLoadNV` and `OpCooperativeMatrixStoreNV` **may** not be bounds-checked.

**Note**



If a SPIR-V `OpLoad` instruction loads a structure and the tail end of the structure is out of bounds, then all members of the structure are considered out of bounds even if the members at the end are not statically used.

- If any buffer access is determined to be out of bounds, then any other access of the same type (load, store, or atomic) to the same buffer that accesses an address less than 16 bytes away from the out of bounds address **may** also be considered out of bounds.
- Out-of-bounds buffer loads will return any of the following values:
  - Values from anywhere within the memory range(s) bound to the buffer (possibly including bytes of memory past the end of the buffer, up to the end of the bound range).
  - Zero values, or (0,0,0,x) vectors for vector reads where x is a valid value represented in the type of the vector components and **may** be any of:
    - 0, 1, or the maximum representable positive integer value, for signed or unsigned integer components
    - 0.0 or 1.0, for floating-point components
- Out-of-bounds writes **may** modify values within the memory range(s) bound to the buffer, but **must** not modify any other memory.
- Out-of-bounds atomics **may** modify values within the memory range(s) bound to the buffer, but **must** not modify any other memory, and return an undefined value.
- Vertex input attributes are considered out of bounds if the offset of the attribute in the bound vertex buffer range plus the size of the attribute is greater than either:
  - `vertexBufferRangeSize`, if `bindingStride` == 0; or
  - (`vertexBufferRangeSize` - (`vertexBufferRangeSize` % `bindingStride`))

where `vertexBufferRangeSize` is the byte size of the memory range bound to the vertex buffer binding and `bindingStride` is the byte stride of the corresponding vertex input binding. Further, if any vertex input attribute using a specific vertex input binding is out of bounds, then all vertex input attributes using that vertex input binding for that vertex shader invocation are considered out of bounds.

- If a vertex input attribute is out of bounds, it will be assigned one of the following values:
  - Values from anywhere within the memory range(s) bound to the buffer, converted according to the format of the attribute.
  - Zero values, format converted according to the format of the attribute.
  - Zero values, or (0,0,0,x) vectors, as described above.
- If `robustBufferAccess` is not enabled, applications **must** not perform out of bounds accesses.
- `fullDrawIndexUInt32` specifies the full 32-bit range of indices is supported for indexed draw calls when using a `VkIndexType` of `VK_INDEX_TYPE_UINT32`. `maxDrawIndexedIndexValue` is the maximum index value that **may** be used (aside from the primitive restart index, which is always  $2^{32}-1$ )

when the `VkIndexType` is `VK_INDEX_TYPE_UINT32`). If this feature is supported, `maxDrawIndexedIndexValue` **must** be  $2^{32}-1$ ; otherwise it **must** be no smaller than  $2^{24}-1$ . See `maxDrawIndexedIndexValue`.

- `imageCubeArray` specifies whether image views with a `VkImageViewType` of `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY` **can** be created, and that the corresponding `SampledCubeArray` and `ImageCubeArray` SPIR-V capabilities **can** be used in shader code.
- `independentBlend` specifies whether the `VkPipelineColorBlendAttachmentState` settings are controlled independently per-attachment. If this feature is not enabled, the `VkPipelineColorBlendAttachmentState` settings for all color attachments **must** be identical. Otherwise, a different `VkPipelineColorBlendAttachmentState` **can** be provided for each bound color attachment.
- `geometryShader` specifies whether geometry shaders are supported. If this feature is not enabled, the `VK_SHADER_STAGE_GEOMETRY_BIT` and `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT` enum values **must** not be used. This also specifies whether shader modules **can** declare the `Geometry` capability.
- `tessellationShader` specifies whether tessellation control and evaluation shaders are supported. If this feature is not enabled, the `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT`, `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`, `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`, and `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO` enum values **must** not be used. This also specifies whether shader modules **can** declare the `Tessellation` capability.
- `sampleRateShading` specifies whether `Sample Shading` and multisample interpolation are supported. If this feature is not enabled, the `sampleShadingEnable` member of the `VkPipelineMultisampleStateCreateInfo` structure **must** be set to `VK_FALSE` and the `minSampleShading` member is ignored. This also specifies whether shader modules **can** declare the `SampleRateShading` capability.
- `dualSrcBlend` specifies whether blend operations which take two sources are supported. If this feature is not enabled, the `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, and `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA` enum values **must** not be used as source or destination blending factors. See [Dual-Source Blending](#).
- `logicOp` specifies whether logic operations are supported. If this feature is not enabled, the `logicOpEnable` member of the `VkPipelineColorBlendStateCreateInfo` structure **must** be set to `VK_FALSE`, and the `logicOp` member is ignored.
- `multiDrawIndirect` specifies whether multiple draw indirect is supported. If this feature is not enabled, the `drawCount` parameter to the `vkCmdDrawIndirect` and `vkCmdDrawIndexedIndirect` commands **must** be 0 or 1. The `maxDrawIndirectCount` member of the `VkPhysicalDeviceLimits` structure **must** also be 1 if this feature is not supported. See `maxDrawIndirectCount`.
- `drawIndirectFirstInstance` specifies whether indirect draw calls support the `firstInstance` parameter. If this feature is not enabled, the `firstInstance` member of all `VkDrawIndirectCommand` and `VkDrawIndexedIndirectCommand` structures that are provided to the `vkCmdDrawIndirect` and `vkCmdDrawIndexedIndirect` commands **must** be 0.
- `depthClamp` specifies whether depth clamping is supported. If this feature is not enabled, the `depthClampEnable` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to `VK_FALSE`. Otherwise, setting `depthClampEnable` to `VK_TRUE` will enable depth clamping.

- `depthBiasClamp` specifies whether depth bias clamping is supported. If this feature is not enabled, the `depthBiasClamp` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to 0.0 unless the `VK_DYNAMIC_STATE_DEPTH_BIAS` dynamic state is enabled, and the `depthBiasClamp` parameter to `vkCmdSetDepthBias` **must** be set to 0.0.
- `fillModeNonSolid` specifies whether point and wireframe fill modes are supported. If this feature is not enabled, the `VK_POLYGON_MODE_POINT` and `VK_POLYGON_MODE_LINE` enum values **must** not be used.
- `depthBounds` specifies whether depth bounds tests are supported. If this feature is not enabled, the `depthBoundsTestEnable` member of the `VkPipelineDepthStencilStateCreateInfo` structure **must** be set to `VK_FALSE`. When `depthBoundsTestEnable` is set to `VK_FALSE`, the `minDepthBounds` and `maxDepthBounds` members of the `VkPipelineDepthStencilStateCreateInfo` structure are ignored.
- `wideLines` specifies whether lines with width other than 1.0 are supported. If this feature is not enabled, the `lineWidth` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to 1.0 unless the `VK_DYNAMIC_STATE_LINE_WIDTH` dynamic state is enabled, and the `lineWidth` parameter to `vkCmdSetLineWidth` **must** be set to 1.0. When this feature is supported, the range and granularity of supported line widths are indicated by the `lineWidthRange` and `lineWidthGranularity` members of the `VkPhysicalDeviceLimits` structure, respectively.
- `largePoints` specifies whether points with size greater than 1.0 are supported. If this feature is not enabled, only a point size of 1.0 written by a shader is supported. The range and granularity of supported point sizes are indicated by the `pointSizeRange` and `pointSizeGranularity` members of the `VkPhysicalDeviceLimits` structure, respectively.
- `alphaToOne` specifies whether the implementation is able to replace the alpha value of the color fragment output from the fragment shader with the maximum representable alpha value for fixed-point colors or 1.0 for floating-point colors. If this feature is not enabled, then the `alphaToOneEnable` member of the `VkPipelineMultisampleStateCreateInfo` structure **must** be set to `VK_FALSE`. Otherwise setting `alphaToOneEnable` to `VK_TRUE` will enable alpha-to-one behavior.
- `multiViewport` specifies whether more than one viewport is supported. If this feature is not enabled:
  - The `viewportCount` and `scissorCount` members of the `VkPipelineViewportStateCreateInfo` structure **must** be set to 1.
  - The `firstViewport` and `viewportCount` parameters to the `vkCmdSetViewport` command **must** be set to 0 and 1, respectively.
  - The `firstScissor` and `scissorCount` parameters to the `vkCmdSetScissor` command **must** be set to 0 and 1, respectively.
  - The `exclusiveScissorCount` member of the `VkPipelineViewportExclusiveScissorStateCreateInfoNV` structure **must** be set to 0 or 1.
  - The `firstExclusiveScissor` and `exclusiveScissorCount` parameters to the `vkCmdSetExclusiveScissorNV` command **must** be set to 0 and 1, respectively.
- `samplerAnisotropy` specifies whether anisotropic filtering is supported. If this feature is not enabled, the `anisotropyEnable` member of the `VkSamplerCreateInfo` structure **must** be `VK_FALSE`.
- `textureCompressionETC2` specifies whether all of the ETC2 and EAC compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

features **must** be supported in `optimalTilingFeatures` for the following formats:

- `VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK`
- `VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK`
- `VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK`
- `VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK`
- `VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK`
- `VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK`
- `VK_FORMAT_EAC_R11_UNORM_BLOCK`
- `VK_FORMAT_EAC_R11_SNORM_BLOCK`
- `VK_FORMAT_EAC_R11G11_UNORM_BLOCK`
- `VK_FORMAT_EAC_R11G11_SNORM_BLOCK`

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` **can** be used to check for supported properties of individual formats as normal.

- `textureCompressionASTC_LDR` specifies whether all of the ASTC LDR compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features **must** be supported in `optimalTilingFeatures` for the following formats:

- `VK_FORMAT_ASTC_4x4_UNORM_BLOCK`
- `VK_FORMAT_ASTC_4x4_SRGB_BLOCK`
- `VK_FORMAT_ASTC_5x4_UNORM_BLOCK`
- `VK_FORMAT_ASTC_5x4_SRGB_BLOCK`
- `VK_FORMAT_ASTC_5x5_UNORM_BLOCK`
- `VK_FORMAT_ASTC_5x5_SRGB_BLOCK`
- `VK_FORMAT_ASTC_6x5_UNORM_BLOCK`
- `VK_FORMAT_ASTC_6x5_SRGB_BLOCK`
- `VK_FORMAT_ASTC_6x6_UNORM_BLOCK`
- `VK_FORMAT_ASTC_6x6_SRGB_BLOCK`
- `VK_FORMAT_ASTC_8x5_UNORM_BLOCK`
- `VK_FORMAT_ASTC_8x5_SRGB_BLOCK`
- `VK_FORMAT_ASTC_8x6_UNORM_BLOCK`
- `VK_FORMAT_ASTC_8x6_SRGB_BLOCK`
- `VK_FORMAT_ASTC_8x8_UNORM_BLOCK`
- `VK_FORMAT_ASTC_8x8_SRGB_BLOCK`
- `VK_FORMAT_ASTC_10x5_UNORM_BLOCK`
- `VK_FORMAT_ASTC_10x5_SRGB_BLOCK`
- `VK_FORMAT_ASTC_10x6_UNORM_BLOCK`
- `VK_FORMAT_ASTC_10x6_SRGB_BLOCK`
- `VK_FORMAT_ASTC_10x8_UNORM_BLOCK`
- `VK_FORMAT_ASTC_10x8_SRGB_BLOCK`
- `VK_FORMAT_ASTC_10x10_UNORM_BLOCK`

- `VK_FORMAT_ASTC_10x10_SRGB_BLOCK`
- `VK_FORMAT_ASTC_12x10_UNORM_BLOCK`
- `VK_FORMAT_ASTC_12x10_SRGB_BLOCK`
- `VK_FORMAT_ASTC_12x12_UNORM_BLOCK`
- `VK_FORMAT_ASTC_12x12_SRGB_BLOCK`

To query for additional properties, or if the feature is not enabled, [vkGetPhysicalDeviceFormatProperties](#) and [vkGetPhysicalDeviceImageFormatProperties](#) **can** be used to check for supported properties of individual formats as normal.

- `textureCompressionBC` specifies whether all of the BC compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features **must** be supported in `optimalTilingFeatures` for the following formats:

- `VK_FORMAT_BC1_RGB_UNORM_BLOCK`
- `VK_FORMAT_BC1_RGB_SRGB_BLOCK`
- `VK_FORMAT_BC1_RGBA_UNORM_BLOCK`
- `VK_FORMAT_BC1_RGBA_SRGB_BLOCK`
- `VK_FORMAT_BC2_UNORM_BLOCK`
- `VK_FORMAT_BC2_SRGB_BLOCK`
- `VK_FORMAT_BC3_UNORM_BLOCK`
- `VK_FORMAT_BC3_SRGB_BLOCK`
- `VK_FORMAT_BC4_UNORM_BLOCK`
- `VK_FORMAT_BC4_SNORM_BLOCK`
- `VK_FORMAT_BC5_UNORM_BLOCK`
- `VK_FORMAT_BC5_SNORM_BLOCK`
- `VK_FORMAT_BC6H_UFLOAT_BLOCK`
- `VK_FORMAT_BC6H_SFLOAT_BLOCK`
- `VK_FORMAT_BC7_UNORM_BLOCK`
- `VK_FORMAT_BC7_SRGB_BLOCK`

To query for additional properties, or if the feature is not enabled, [vkGetPhysicalDeviceFormatProperties](#) and [vkGetPhysicalDeviceImageFormatProperties](#) **can** be used to check for supported properties of individual formats as normal.

- `occlusionQueryPrecise` specifies whether occlusion queries returning actual sample counts are supported. Occlusion queries are created in a `VkQueryPool` by specifying the `queryType` of `VK_QUERY_TYPE_OCCLUSION` in the `VkQueryPoolCreateInfo` structure which is passed to `vkCreateQueryPool`. If this feature is enabled, queries of this type **can** enable `VK_QUERY_CONTROL_PRECISE_BIT` in the `flags` parameter to `vkCmdBeginQuery`. If this feature is not supported, the implementation supports only boolean occlusion queries. When any samples are passed, boolean queries will return a non-zero result value, otherwise a result value of zero is returned. When this feature is enabled and `VK_QUERY_CONTROL_PRECISE_BIT` is set, occlusion queries will report the actual number of samples passed.
- `pipelineStatisticsQuery` specifies whether the pipeline statistics queries are supported. If this feature is not enabled, queries of type `VK_QUERY_TYPE_PIPELINE_STATISTICS` **cannot** be created,

and none of the `VkQueryPipelineStatisticFlagBits` bits **can** be set in the `pipelineStatistics` member of the `VkQueryPoolCreateInfo` structure.

- `vertexPipelineStoresAndAtomics` specifies whether storage buffers and images support stores and atomic operations in the vertex, tessellation, and geometry shader stages. If this feature is not enabled, all storage image, storage texel buffers, and storage buffer variables used by these stages in shader modules **must** be decorated with the `NonWritable` decoration (or the `readonly` memory qualifier in GLSL).
- `fragmentStoresAndAtomics` specifies whether storage buffers and images support stores and atomic operations in the fragment shader stage. If this feature is not enabled, all storage image, storage texel buffers, and storage buffer variables used by the fragment stage in shader modules **must** be decorated with the `NonWritable` decoration (or the `readonly` memory qualifier in GLSL).
- `shaderTessellationAndGeometryPointSize` specifies whether the `PointSize` built-in decoration is available in the tessellation control, tessellation evaluation, and geometry shader stages. If this feature is not enabled, members decorated with the `PointSize` built-in decoration **must** not be read from or written to and all points written from a tessellation or geometry shader will have a size of 1.0. This also specifies whether shader modules **can** declare the `TessellationPointSize` capability for tessellation control and evaluation shaders, or if the shader modules **can** declare the `GeometryPointSize` capability for geometry shaders. An implementation supporting this feature **must** also support one or both of the `tessellationShader` or `geometryShader` features.
- `shaderImageGatherExtended` specifies whether the extended set of image gather instructions are available in shader code. If this feature is not enabled, the `OpImage*Gather` instructions do not support the `Offset` and `ConstOffsets` operands. This also specifies whether shader modules **can** declare the `ImageGatherExtended` capability.
- `shaderStorageImageExtendedFormats` specifies whether all the extended storage image formats are available in shader code. If this feature is enabled then the `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT` feature **must** be supported in `optimalTilingFeatures` for all of the extended formats. To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` **can** be used to check for supported properties of individual formats as normal.
- `shaderStorageImageMultisample` specifies whether multisampled storage images are supported. If this feature is not enabled, images that are created with a `usage` that includes `VK_IMAGE_USAGE_STORAGE_BIT` **must** be created with `samples` equal to `VK_SAMPLE_COUNT_1_BIT`. This also specifies whether shader modules **can** declare the `StorageImageMultisample` capability.
- `shaderStorageImageReadWithoutFormat` specifies whether storage images require a format qualifier to be specified when reading from storage images. If this feature is not enabled, the `OpImageRead` instruction **must** not have an `OpTypeImage` of `Unknown`. This also specifies whether shader modules **can** declare the `StorageImageReadWithoutFormat` capability.
- `shaderStorageImageWriteWithoutFormat` specifies whether storage images require a format qualifier to be specified when writing to storage images. If this feature is not enabled, the `OpImageWrite` instruction **must** not have an `OpTypeImage` of `Unknown`. This also specifies whether shader modules **can** declare the `StorageImageWriteWithoutFormat` capability.
- `shaderUniformBufferArrayDynamicIndexing` specifies whether arrays of uniform buffers **can** be indexed by *dynamically uniform* integer expressions in shader code. If this feature is not

enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` **must** be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules **can** declare the `UniformBufferArrayDynamicIndexing` capability.

- `shaderSampledImageArrayDynamicIndexing` specifies whether arrays of samplers or sampled images **can** be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` **must** be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules **can** declare the `SampledImageArrayDynamicIndexing` capability.
- `shaderStorageBufferArrayDynamicIndexing` specifies whether arrays of storage buffers **can** be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` **must** be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules **can** declare the `StorageBufferArrayDynamicIndexing` capability.
- `shaderStorageImageArrayDynamicIndexing` specifies whether arrays of storage images **can** be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` **must** be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules **can** declare the `StorageImageArrayDynamicIndexing` capability.
- `shaderClipDistance` specifies whether clip distances are supported in shader code. If this feature is not enabled, any members decorated with the `ClipDistance` built-in decoration **must** not be read from or written to in shader modules. This also specifies whether shader modules **can** declare the `ClipDistance` capability.
- `shaderCullDistance` specifies whether cull distances are supported in shader code. If this feature is not enabled, any members decorated with the `CullDistance` built-in decoration **must** not be read from or written to in shader modules. This also specifies whether shader modules **can** declare the `CullDistance` capability.
- `shaderFloat64` specifies whether 64-bit floats (doubles) are supported in shader code. If this feature is not enabled, 64-bit floating-point types **must** not be used in shader code. This also specifies whether shader modules **can** declare the `Float64` capability. Declaring and using 64-bit floats is enabled for all storage classes that SPIR-V allows with the `Float64` capability.
- `shaderInt64` specifies whether 64-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 64-bit integer types **must** not be used in shader code. This also specifies whether shader modules **can** declare the `Int64` capability. Declaring and using 64-bit integers is enabled for all storage classes that SPIR-V allows with the `Int64` capability.
- `shaderInt16` specifies whether 16-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 16-bit integer types **must** not be used in shader code. This also specifies whether shader modules **can** declare the `Int16` capability. However, this only enables a subset of the storage classes that SPIR-V allows for the `Int16` SPIR-V capability: Declaring and using 16-bit integers in the `Private`, `Workgroup`, and `Function` storage classes is enabled, while declaring them in the interface storage classes (e.g., `UniformConstant`, `Uniform`, `StorageBuffer`, `Input`, `Output`, and `PushConstant`) is not enabled.

- `shaderResourceResidency` specifies whether image operations that return resource residency information are supported in shader code. If this feature is not enabled, the `OpImageSparse*` instructions **must** not be used in shader code. This also specifies whether shader modules **can** declare the `SparseResidency` capability. The feature requires at least one of the `sparseResidency*` features to be supported.
- `shaderResourceMinLod` specifies whether image operations specifying the minimum resource LOD are supported in shader code. If this feature is not enabled, the `MinLod` image operand **must** not be used in shader code. This also specifies whether shader modules **can** declare the `MinLod` capability.
- `sparseBinding` specifies whether resource memory **can** be managed at opaque sparse block level instead of at the object level. If this feature is not enabled, resource memory **must** be bound only on a per-object basis using the `vkBindBufferMemory` and `vkBindImageMemory` commands. In this case, buffers and images **must** not be created with `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` and `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` set in the `flags` member of the `VkBufferCreateInfo` and `VkImageCreateInfo` structures, respectively. Otherwise resource memory **can** be managed as described in [Sparse Resource Features](#).
- `sparseResidencyBuffer` specifies whether the device **can** access partially resident buffers. If this feature is not enabled, buffers **must** not be created with `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` set in the `flags` member of the `VkBufferCreateInfo` structure.
- `sparseResidencyImage2D` specifies whether the device **can** access partially resident 2D images with 1 sample per pixel. If this feature is not enabled, images with an `imageType` of `VK_IMAGE_TYPE_2D` and `samples` set to `VK_SAMPLE_COUNT_1_BIT` **must** not be created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set in the `flags` member of the `VkImageCreateInfo` structure.
- `sparseResidencyImage3D` specifies whether the device **can** access partially resident 3D images. If this feature is not enabled, images with an `imageType` of `VK_IMAGE_TYPE_3D` **must** not be created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set in the `flags` member of the `VkImageCreateInfo` structure.
- `sparseResidency2Samples` specifies whether the physical device **can** access partially resident 2D images with 2 samples per pixel. If this feature is not enabled, images with an `imageType` of `VK_IMAGE_TYPE_2D` and `samples` set to `VK_SAMPLE_COUNT_2_BIT` **must** not be created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set in the `flags` member of the `VkImageCreateInfo` structure.
- `sparseResidency4Samples` specifies whether the physical device **can** access partially resident 2D images with 4 samples per pixel. If this feature is not enabled, images with an `imageType` of `VK_IMAGE_TYPE_2D` and `samples` set to `VK_SAMPLE_COUNT_4_BIT` **must** not be created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set in the `flags` member of the `VkImageCreateInfo` structure.
- `sparseResidency8Samples` specifies whether the physical device **can** access partially resident 2D images with 8 samples per pixel. If this feature is not enabled, images with an `imageType` of `VK_IMAGE_TYPE_2D` and `samples` set to `VK_SAMPLE_COUNT_8_BIT` **must** not be created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set in the `flags` member of the `VkImageCreateInfo` structure.
- `sparseResidency16Samples` specifies whether the physical device **can** access partially resident 2D

images with 16 samples per pixel. If this feature is not enabled, images with an `imageType` of `VK_IMAGE_TYPE_2D` and `samples` set to `VK_SAMPLE_COUNT_16_BIT` **must** not be created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set in the `flags` member of the `VkImageCreateInfo` structure.

- `sparseResidencyAliased` specifies whether the physical device **can** correctly access data aliased into multiple locations. If this feature is not enabled, the `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` and `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT` enum values **must** not be used in `flags` members of the `VkBufferCreateInfo` and `VkImageCreateInfo` structures, respectively.
- `variableMultisampleRate` specifies whether all pipelines that will be bound to a command buffer during a subpass with no attachments **must** have the same value for `VkPipelineMultisampleStateCreateInfo::rasterizationSamples`. If set to `VK_TRUE`, the implementation supports variable multisample rates in a subpass with no attachments. If set to `VK_FALSE`, then all pipelines bound in such a subpass **must** have the same multisample rate. This has no effect in situations where a subpass uses any attachments.
- `inheritedQueries` specifies whether a secondary command buffer **may** be executed while a query is active.

The `VkPhysicalDeviceVariablePointersFeatures` structure is defined as:

```
typedef struct VkPhysicalDeviceVariablePointersFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           variablePointersStorageBuffer;
    VkBool32           variablePointers;
} VkPhysicalDeviceVariablePointersFeatures;
```

or the equivalent

```
typedef VkPhysicalDeviceVariablePointersFeatures
VkPhysicalDeviceVariablePointersFeaturesKHR;
```

The members of the `VkPhysicalDeviceVariablePointersFeatures` structure describe the following features:

- `variablePointersStorageBuffer` specifies whether the implementation supports the SPIR-V `VariablePointersStorageBuffer` capability. When this feature is not enabled, shader modules **must** not declare the `SPV_KHR_variable_pointers` extension or the `VariablePointersStorageBuffer` capability.
- `variablePointers` specifies whether the implementation supports the SPIR-V `VariablePointers` capability. When this feature is not enabled, shader modules **must** not declare the `VariablePointers` capability.

If the `VkPhysicalDeviceVariablePointersFeatures` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceVariablePointersFeatures` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the features.

## Valid Usage

- If `variablePointers` is enabled then `variablePointersStorageBuffer` **must** also be enabled.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES`

The `VkPhysicalDeviceMultiviewFeatures` structure is defined as:

```
typedef struct VkPhysicalDeviceMultiviewFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            multiview;
    VkBool32            multiviewGeometryShader;
    VkBool32            multiviewTessellationShader;
} VkPhysicalDeviceMultiviewFeatures;
```

or the equivalent

```
typedef VkPhysicalDeviceMultiviewFeatures VkPhysicalDeviceMultiviewFeaturesKHR;
```

The members of the `VkPhysicalDeviceMultiviewFeatures` structure describe the following features:

- `multiview` specifies whether the implementation supports multiview rendering within a render pass. If this feature is not enabled, the view mask of each subpass **must** always be zero.
- `multiviewGeometryShader` specifies whether the implementation supports multiview rendering within a render pass, with `geometry shaders`. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask **must** not include a geometry shader.
- `multiviewTessellationShader` specifies whether the implementation supports multiview rendering within a render pass, with `tessellation shaders`. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask **must** not include any tessellation shaders.

If the `VkPhysicalDeviceMultiviewFeatures` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceMultiviewFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the features.

## Valid Usage

- If `multiviewGeometryShader` is enabled then `multiview` **must** also be enabled.
- If `multiviewTessellationShader` is enabled then `multiview` **must** also be enabled.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES`

To query 64-bit atomic support for signed and unsigned integers call `vkGetPhysicalDeviceFeatures2` with a `VkPhysicalDeviceShaderAtomicInt64FeaturesKHR` structure included in the `pNext` chain of its `pFeatures` parameter.

The `VkPhysicalDeviceShaderAtomicInt64FeaturesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderAtomicInt64FeaturesKHR {  
    VkStructureType    sType;  
    void*             pNext;  
    VkBool32          shaderBufferInt64Atomics;  
    VkBool32          shaderSharedInt64Atomics;  
} VkPhysicalDeviceShaderAtomicInt64FeaturesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shaderBufferInt64Atomics` indicates whether shaders **can** support 64-bit unsigned and signed integer atomic operations on buffers.
- `shaderSharedInt64Atomics` indicates whether shaders **can** support 64-bit unsigned and signed integer atomic operations on shared memory.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES_KHR`

To query 8-bit storage features additionally supported call `vkGetPhysicalDeviceFeatures2` with a `VkPhysicalDevice8BitStorageFeaturesKHR` structure included in the `pNext` chain of its `pFeatures` parameter. The `VkPhysicalDevice8BitStorageFeaturesKHR` structure **can** also be in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which additional features are enabled in the device.

The `VkPhysicalDevice8BitStorageFeaturesKHR` structure is defined as:

```

typedef struct VkPhysicalDevice8BitStorageFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer8BitAccess;
    VkBool32 uniformAndStorageBuffer8BitAccess;
    VkBool32 storagePushConstant8;
} VkPhysicalDevice8BitStorageFeaturesKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **storageBuffer8BitAccess** indicates whether objects in the **StorageBuffer** or **PhysicalStorageBuffer** storage class with the **Block** decoration **can** have 8-bit integer members. If this feature is not enabled, 8-bit integer members **must** not be used in such objects. This also indicates whether shader modules **can** declare the **StorageBuffer8BitAccess** capability.
- **uniformAndStorageBuffer8BitAccess** indicates whether objects in the **Uniform** storage class with the **Block** decoration and in the **StorageBuffer** or **PhysicalStorageBuffer** storage class with the same decoration **can** have 8-bit integer members. If this feature is not enabled, 8-bit integer members **must** not be used in such objects. This also indicates whether shader modules **can** declare the **UniformAndStorageBuffer8BitAccess** capability.
- **storagePushConstant8** indicates whether objects in the **PushConstant** storage class **can** have 8-bit integer members. If this feature is not enabled, 8-bit integer members **must** not be used in such objects. This also indicates whether shader modules **can** declare the **StoragePushConstant8** capability.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_8BIT\_STORAGE\_FEATURES\_KHR**

To query 16-bit storage features additionally supported call `vkGetPhysicalDeviceFeatures2` with a `VkPhysicalDevice16BitStorageFeatures` structure included in the `pNext` chain of its `pFeatures` parameter. The `VkPhysicalDevice16BitStorageFeatures` structure **can** also be in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which additional features are enabled in the device.

The `VkPhysicalDevice16BitStorageFeatures` structure is defined as:

```

typedef struct VkPhysicalDevice16BitStorageFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 storageBuffer16BitAccess;
    VkBool32 uniformAndStorageBuffer16BitAccess;
    VkBool32 storagePushConstant16;
    VkBool32 storageInputOutput16;
} VkPhysicalDevice16BitStorageFeatures;

```

or the equivalent

```
typedef VkPhysicalDevice16BitStorageFeatures VkPhysicalDevice16BitStorageFeaturesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `storageBuffer16BitAccess` specifies whether objects in the `StorageBuffer` or `PhysicalStorageBuffer` storage class with the `Block` decoration **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `StorageBuffer16BitAccess` capability.
- `uniformAndStorageBuffer16BitAccess` specifies whether objects in the `Uniform` storage class with the `Block` decoration and in the `StorageBuffer` or `PhysicalStorageBuffer` storage class with the same decoration **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `UniformAndStorageBuffer16BitAccess` capability.
- `storagePushConstant16` specifies whether objects in the `PushConstant` storage class **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `StoragePushConstant16` capability.
- `storageInputOutput16` specifies whether objects in the `Input` and `Output` storage classes **can** have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members **must** not be used in such objects. This also specifies whether shader modules **can** declare the `StorageInputOutput16` capability.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES`

To query features additionally supported by the `VK_KHR_shader_float16_int8` extension, call `vkGetPhysicalDeviceFeatures2KHR` with a `VkPhysicalDeviceShaderFloat16Int8FeaturesKHR` structure in the `pNext` chain. The `VkPhysicalDeviceShaderFloat16Int8FeaturesKHR` structure **can** also be in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which additional features are enabled in the device.

The `VkPhysicalDeviceShaderFloat16Int8FeaturesKHR` structure is defined as:

```

typedef struct VkPhysicalDeviceShaderFloat16Int8FeaturesKHR {
    VkStructureType      sType;
    void*              pNext;
    VkBool32             shaderFloat16;
    VkBool32             shaderInt8;
} VkPhysicalDeviceShaderFloat16Int8FeaturesKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **shaderFloat16** indicates whether 16-bit floats (halfs) are supported in shader code. This also indicates whether shader modules **can** declare the **Float16** capability. However, this only enables a subset of the storage classes that SPIR-V allows for the **Float16** SPIR-V capability: Declaring and using 16-bit floats in the **Private**, **Workgroup**, and **Function** storage classes is enabled, while declaring them in the interface storage classes (e.g., **UniformConstant**, **Uniform**, **StorageBuffer**, **Input**, **Output**, and **PushConstant**) is not enabled.
- **shaderInt8** indicates whether 8-bit integers (signed and unsigned) are supported in shader code. This also indicates whether shader modules **can** declare the **Int8** capability. However, this only enables a subset of the storage classes that SPIR-V allows for the **Int8** SPIR-V capability: Declaring and using 8-bit integers in the **Private**, **Workgroup**, and **Function** storage classes is enabled, while declaring them in the interface storage classes (e.g., **UniformConstant**, **Uniform**, **StorageBuffer**, **Input**, **Output**, and **PushConstant**) is not enabled.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_SHADER\_FLOAT16\_INT8\_FEATURES\_KHR**

To query shader clock support, call **vkGetPhysicalDeviceFeatures2** with a **VkPhysicalDeviceShaderClockFeaturesKHR** structure included in the **pNext** chain of its **pFeatures** parameter.

The **VkPhysicalDeviceShaderClockFeaturesKHR** structure is defined as:

```

typedef struct VkPhysicalDeviceShaderClockFeaturesKHR {
    VkStructureType      sType;
    void*              pNext;
    VkBool32             shaderSubgroupClock;
    VkBool32             shaderDeviceClock;
} VkPhysicalDeviceShaderClockFeaturesKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **shaderSubgroupClock** indicates whether shaders **can** support **Subgroup** scoped clock reads.
- **shaderDeviceClock** indicates whether shaders **can** support **Device** scoped clock reads.

If the `VkPhysicalDeviceShaderClockFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceShaderClockFeaturesKHR` can also be used in `pNext` chain of `VkDeviceCreateInfo` to enable the features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CLOCK_FEATURES_KHR`

The `VkPhysicalDeviceSamplerYcbcrConversionFeatures` structure is defined as:

```
typedef struct VkPhysicalDeviceSamplerYcbcrConversionFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            samplerYcbcrConversion;
} VkPhysicalDeviceSamplerYcbcrConversionFeatures;
```

or the equivalent

```
typedef VkPhysicalDeviceSamplerYcbcrConversionFeatures
VkPhysicalDeviceSamplerYcbcrConversionFeaturesKHR;
```

The members of the `VkPhysicalDeviceSamplerYcbcrConversionFeatures` structure describe the following feature:

- `samplerYcbcrConversion` specifies whether the implementation supports `sampler Y'CbCr conversion`. If `samplerYcbcrConversion` is `VK_FALSE`, sampler Y'CbCr conversion is not supported, and samplers using sampler Y'CbCr conversion must not be used.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES`

The `VkPhysicalDeviceProtectedMemoryFeatures` structure is defined as:

```
typedef struct VkPhysicalDeviceProtectedMemoryFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            protectedMemory;
} VkPhysicalDeviceProtectedMemoryFeatures;
```

- `protectedMemory` specifies whether protected memory is supported.

If the `VkPhysicalDeviceProtectedMemoryFeatures` structure is included in the `pNext` chain of

`VkPhysicalDeviceFeatures2`, it is filled with a value indicating whether the feature is supported.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES`

The `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            advancedBlendCoherentOperations;
} VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT;
```

The members of the `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` structure describe the following features:

- `advancedBlendCoherentOperations` specifies whether blending using `advanced blend operations` is guaranteed to execute atomically and in `primitive order`. If this is `VK_TRUE`, `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT` is treated the same as `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`, and advanced blending needs no additional synchronization over basic blending. If this is `VK_FALSE`, then memory dependencies are required to guarantee order between two advanced blending operations that occur on the same sample.

If the `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` can also be used in `pNext` chain of `VkDeviceCreateInfo` to enable the features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_FEATURES_EXT`

The `VkPhysicalDeviceConditionalRenderingFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceConditionalRenderingFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            conditionalRendering;
    VkBool32            inheritedConditionalRendering;
} VkPhysicalDeviceConditionalRenderingFeaturesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `conditionalRendering` specifies whether conditional rendering is supported.
- `inheritedConditionalRendering` specifies whether a secondary command buffer **can** be executed while conditional rendering is active in the primary command buffer.

If the `VkPhysicalDeviceConditionalRenderingFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating the implementation-dependent behavior. `VkPhysicalDeviceConditionalRenderingFeaturesEXT` **can** also be used in `pNext` chain of `VkDeviceCreateInfo` to enable the features.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONDITIONAL_RENDERING_FEATURES_EXT`

The `VkPhysicalDeviceShaderDrawParametersFeatures` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderDrawParametersFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            shaderDrawParameters;
} VkPhysicalDeviceShaderDrawParametersFeatures;
```

- `shaderDrawParameters` specifies whether shader draw parameters are supported.

If the `VkPhysicalDeviceShaderDrawParametersFeatures` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with a value indicating whether the feature is supported.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES`

The `VkPhysicalDeviceMeshShaderFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceMeshShaderFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            taskShader;
    VkBool32            meshShader;
} VkPhysicalDeviceMeshShaderFeaturesNV;
```

- `taskShader` indicates whether the task shader stage is supported.
- `meshShader` indicates whether the mesh shader stage is supported.

If the `VkPhysicalDeviceMeshShaderFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with a value indicating whether the feature is supported. `VkPhysicalDeviceMeshShaderFeaturesNV` **can** also be used in `pNext` chain of `VkDeviceCreateInfo` to

enable the features.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_FEATURES_NV`

The `VkPhysicalDeviceDescriptorIndexingFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceDescriptorIndexingFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            shaderInputAttachmentArrayDynamicIndexing;
    VkBool32            shaderUniformTexelBufferArrayDynamicIndexing;
    VkBool32            shaderStorageTexelBufferArrayDynamicIndexing;
    VkBool32            shaderUniformBufferArrayNonUniformIndexing;
    VkBool32            shaderSampledImageArrayNonUniformIndexing;
    VkBool32            shaderStorageBufferArrayNonUniformIndexing;
    VkBool32            shaderStorageImageArrayNonUniformIndexing;
    VkBool32            shaderInputAttachmentArrayNonUniformIndexing;
    VkBool32            shaderUniformTexelBufferArrayNonUniformIndexing;
    VkBool32            shaderStorageTexelBufferArrayNonUniformIndexing;
    VkBool32            descriptorBindingUniformBufferUpdateAfterBind;
    VkBool32            descriptorBindingSampledImageUpdateAfterBind;
    VkBool32            descriptorBindingStorageImageUpdateAfterBind;
    VkBool32            descriptorBindingStorageBufferUpdateAfterBind;
    VkBool32            descriptorBindingUniformTexelBufferUpdateAfterBind;
    VkBool32            descriptorBindingStorageTexelBufferUpdateAfterBind;
    VkBool32            descriptorBindingUpdateUnusedWhilePending;
    VkBool32            descriptorBindingPartiallyBound;
    VkBool32            descriptorBindingVariableDescriptorCount;
    VkBool32            runtimeDescriptorArray;
} VkPhysicalDeviceDescriptorIndexingFeaturesEXT;
```

The members of the `VkPhysicalDeviceDescriptorIndexingFeaturesEXT` structure describe the following features:

- `shaderInputAttachmentArrayDynamicIndexing` indicates whether arrays of input attachments **can** be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` **must** be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `InputAttachmentArrayDynamicIndexingEXT` capability.
- `shaderUniformTexelBufferArrayDynamicIndexing` indicates whether arrays of uniform texel buffers **can** be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` **must** be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can**

declare the `UniformTexelBufferArrayDynamicIndexingEXT` capability.

- `shaderStorageTexelBufferArrayDynamicIndexing` indicates whether arrays of storage texel buffers **can** be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` **must** be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `StorageTexelBufferArrayDynamicIndexingEXT` capability.
- `shaderUniformBufferArrayNonUniformIndexing` indicates whether arrays of uniform buffers **can** be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` **must** not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `UniformBufferArrayNonUniformIndexingEXT` capability.
- `shaderSampledImageArrayNonUniformIndexing` indicates whether arrays of samplers or sampled images **can** be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` **must** not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `SampledImageArrayNonUniformIndexingEXT` capability.
- `shaderStorageBufferArrayNonUniformIndexing` indicates whether arrays of storage buffers **can** be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` **must** not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `StorageBufferArrayNonUniformIndexingEXT` capability.
- `shaderStorageImageArrayNonUniformIndexing` indicates whether arrays of storage images **can** be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` **must** not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `StorageImageArrayNonUniformIndexingEXT` capability.
- `shaderInputAttachmentArrayNonUniformIndexing` indicates whether arrays of input attachments **can** be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` **must** not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `InputAttachmentArrayNonUniformIndexingEXT` capability.
- `shaderUniformTexelBufferArrayNonUniformIndexing` indicates whether arrays of uniform texel buffers **can** be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` **must** not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `UniformTexelBufferArrayNonUniformIndexingEXT` capability.
- `shaderStorageTexelBufferArrayNonUniformIndexing` indicates whether arrays of storage texel

buffers **can** be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` **must** not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules **can** declare the `StorageTexelBufferArrayNonUniformIndexingEXT` capability.

- `descriptorBindingUniformBufferUpdateAfterBind` indicates whether the implementation supports updating uniform buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` **must** not be used with `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`.
- `descriptorBindingSampledImageUpdateAfterBind` indicates whether the implementation supports updating sampled image descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` **must** not be used with `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`.
- `descriptorBindingStorageImageUpdateAfterBind` indicates whether the implementation supports updating storage image descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` **must** not be used with `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`.
- `descriptorBindingStorageBufferUpdateAfterBind` indicates whether the implementation supports updating storage buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` **must** not be used with `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`.
- `descriptorBindingUniformTexelBufferUpdateAfterBind` indicates whether the implementation supports updating uniform texel buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` **must** not be used with `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`.
- `descriptorBindingStorageTexelBufferUpdateAfterBind` indicates whether the implementation supports updating storage texel buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT` **must** not be used with `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`.
- `descriptorBindingUpdateUnusedWhilePending` indicates whether the implementation supports updating descriptors while the set is in use. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT` **must** not be used.
- `descriptorBindingPartiallyBound` indicates whether the implementation supports statically using a descriptor set binding in which some descriptors are not valid. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT` **must** not be used.
- `descriptorBindingVariableDescriptorCount` indicates whether the implementation supports descriptor sets with a variable-sized last binding. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT` **must** not be used.
- `runtimeDescriptorArray` indicates whether the implementation supports the SPIR-V `RuntimeDescriptorArrayEXT` capability. If this feature is not enabled, descriptors **must** not be declared in runtime arrays.

If the `VkPhysicalDeviceDescriptorIndexingFeaturesEXT` structure is included in the `pNext` chain of

`VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceDescriptorIndexingFeaturesEXT` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES_EXT`

The `VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           vertexAttributeInstanceRateDivisor;
    VkBool32           vertexAttributeInstanceRateZeroDivisor;
} VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `vertexAttributeInstanceRateDivisor` specifies whether vertex attribute fetching may be repeated in case of instanced rendering.
- `vertexAttributeInstanceRateZeroDivisor` specifies whether a zero value for `VkVertexInputBindingDivisorDescriptionEXT::divisor` is supported.

If the `VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating the implementation-dependent behavior. `VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT` **can** also be used in `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_FEATURES_EXT`

The `VkPhysicalDeviceASTCDecodeFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceASTCDecodeFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           decodeModeSharedExponent;
} VkPhysicalDeviceASTCDecodeFeaturesEXT;
```

The members of the `VkPhysicalDeviceASTCDecodeFeaturesEXT` structure describe the following features:

- `decodeModeSharedExponent` indicates whether the implementation supports decoding ASTC compressed formats to `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32` internal precision.

If the `VkPhysicalDeviceASTCDecodeFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceASTCDecodeFeaturesEXT` **can** also be used in the `pNext` chain of `vkCreateDevice` to enable features.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT`

The `VkPhysicalDeviceTransformFeedbackFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceTransformFeedbackFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            transformFeedback;
    VkBool32            geometryStreams;
} VkPhysicalDeviceTransformFeedbackFeaturesEXT;
```

The members of the `VkPhysicalDeviceTransformFeedbackFeaturesEXT` structure describe the following features:

- `transformFeedback` indicates whether the implementation supports transform feedback and shader modules **can** declare the `TransformFeedback` capability.
- `geometryStreams` indicates whether the implementation supports the `GeometryStreams` SPIR-V capability.

If the `VkPhysicalDeviceTransformFeedbackFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceTransformFeedbackFeaturesEXT` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_FEATURES_EXT`

To query memory model features additionally supported call `vkGetPhysicalDeviceFeatures2` with a `VkPhysicalDeviceVulkanMemoryModelFeaturesKHR` structure included in the `pNext` chain of its `pFeatures` parameter. The `VkPhysicalDeviceVulkanMemoryModelFeaturesKHR` structure **can** also be in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which additional features are enabled in the device.

The `VkPhysicalDeviceVulkanMemoryModelFeaturesKHR` structure is defined as:

```

typedef struct VkPhysicalDeviceVulkanMemoryModelFeaturesKHR {
    VkStructureType      sType;
    void*               pNext;
    VkBool32             vulkanMemoryModel;
    VkBool32             vulkanMemoryModelDeviceScope;
    VkBool32             vulkanMemoryModelAvailabilityVisibilityChains;
} VkPhysicalDeviceVulkanMemoryModelFeaturesKHR;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **vulkanMemoryModel** indicates whether the Vulkan Memory Model is supported, as defined in [Vulkan Memory Model](#). This also indicates whether shader modules **can** declare the [VulkanMemoryModelKHR](#) capability.
- **vulkanMemoryModelDeviceScope** indicates whether the Vulkan Memory Model can use [Device](#) scope synchronization. This also indicates whether shader modules **can** declare the [VulkanMemoryModelDeviceScopeKHR](#) capability.
- **vulkanMemoryModelAvailabilityVisibilityChains** indicates whether the Vulkan Memory Model can use [availability](#) and [visibility chains](#) with more than one element.

## Valid Usage (Implicit)

- **sType** **must** be [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_VULKAN\\_MEMORY\\_MODEL\\_FEATURES\\_KHR](#)

The [VkPhysicalDeviceInlineUniformBlockFeaturesEXT](#) structure is defined as:

```

typedef struct VkPhysicalDeviceInlineUniformBlockFeaturesEXT {
    VkStructureType      sType;
    void*               pNext;
    VkBool32             inlineUniformBlock;
    VkBool32             descriptorBindingInlineUniformBlockUpdateAfterBind;
} VkPhysicalDeviceInlineUniformBlockFeaturesEXT;

```

The members of the [VkPhysicalDeviceInlineUniformBlockFeaturesEXT](#) structure describe the following features:

- **inlineUniformBlock** indicates whether the implementation supports inline uniform block descriptors. If this feature is not enabled, [VK\\_DESCRIPTOR\\_TYPE\\_INLINE\\_UNIFORM\\_BLOCK\\_EXT](#) **must** not be used.
- **descriptorBindingInlineUniformBlockUpdateAfterBind** indicates whether the implementation supports updating inline uniform block descriptors after a set is bound. If this feature is not enabled, [VK\\_DESCRIPTOR\\_BINDING\\_UPDATE\\_AFTER\\_BIND\\_BIT\\_EXT](#) **must** not be used with [VK\\_DESCRIPTOR\\_TYPE\\_INLINE\\_UNIFORM\\_BLOCK\\_EXT](#).

If the [VkPhysicalDeviceInlineUniformBlockFeaturesEXT](#) structure is included in the **pNext** chain of

`VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceInlineUniformBlockFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES_EXT`

The `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          representativeFragmentTest;
} VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV;
```

The members of the `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV` structure describe the following features:

- `representativeFragmentTest` indicates whether the implementation supports the representative fragment test. See [Representative Fragment Test](#).

If the `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_REPRESENTATIVE_FRAGMENT_TEST_FEATURES_NV`

The `VkPhysicalDeviceExclusiveScissorFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceExclusiveScissorFeaturesNV {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          exclusiveScissor;
} VkPhysicalDeviceExclusiveScissorFeaturesNV;
```

The members of the `VkPhysicalDeviceExclusiveScissorFeaturesNV` structure describe the following features:

- `exclusiveScissor` indicates that the implementation supports the exclusive scissor test.

See [Exclusive Scissor Test](#) for more information.

If the `VkPhysicalDeviceExclusiveScissorFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceExclusiveScissorFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXCLUSIVE_SCISSOR_FEATURES_NV`

The `VkPhysicalDeviceCornerSampledImageFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceCornerSampledImageFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            cornerSampledImage;
} VkPhysicalDeviceCornerSampledImageFeaturesNV;
```

The members of the `VkPhysicalDeviceCornerSampledImageFeaturesNV` structure describe the following features:

- `cornerSampledImage` specifies whether images can be created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV`. See [Corner-Sampled Images](#).

If the `VkPhysicalDeviceCornerSampledImageFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceCornerSampledImageFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CORNER_SAMPLED_IMAGE_FEATURES_NV`

The `VkPhysicalDeviceComputeShaderDerivativesFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceComputeShaderDerivativesFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            computeDerivativeGroupQuads;
    VkBool32            computeDerivativeGroupLinear;
} VkPhysicalDeviceComputeShaderDerivativesFeaturesNV;
```

The members of the `VkPhysicalDeviceComputeShaderDerivativesFeaturesNV` structure describe the following features:

- `computeDerivativeGroupQuads` indicates that the implementation supports the

`ComputeDerivativeGroupQuadsNV` SPIR-V capability.

- `computeDerivativeGroupLinear` indicates that the implementation supports the `ComputeDerivativeGroupLinearNV` SPIR-V capability.

See [Compute Shader Derivatives](#) for more information.

If the `VkPhysicalDeviceComputeShaderDerivativesFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceComputeShaderDerivativesFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COMPUTE_SHADER_DERIVATIVES_FEATURES_NV`

The `VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV {  
    VkStructureType    sType;  
    void*             pNext;  
    VkBool32          fragmentShaderBarycentric;  
} VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV;
```

The members of the `VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV` structure describe the following features:

- `fragmentShaderBarycentric` indicates that the implementation supports the `BaryCoordNV` and `BaryCoordNoPerspNV` SPIR-V fragment shader built-ins and supports the `PerVertexNV` SPIR-V decoration on fragment shader input variables.

See [Barycentric Interpolation](#) for more information.

If the `VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_BARYCENTRIC_FEATURES_NV`

The `VkPhysicalDeviceShaderImageFootprintFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderImageFootprintFeaturesNV {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          imageFootprint;
} VkPhysicalDeviceShaderImageFootprintFeaturesNV;
```

- **imageFootprint** specifies whether the implementation supports the **ImageFootprintNV** SPIR-V capability.

See [Texel Footprint Evaluation](#) for more information.

If the **VkPhysicalDeviceShaderImageFootprintFeaturesNV** structure is included in the **pNext** chain of **VkPhysicalDeviceFeatures2**, it is filled with values indicating whether each feature is supported. **VkPhysicalDeviceShaderImageFootprintFeaturesNV** **can** also be used in the **pNext** chain of **VkDeviceCreateInfo** to enable features.

## Valid Usage (Implicit)

- **sType** **must** be **VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_SHADER\_IMAGE\_FOOTPRINT\_FEATURES\_NV**

The **VkPhysicalDeviceShadingRateImageFeaturesNV** structure is defined as:

```
typedef struct VkPhysicalDeviceShadingRateImageFeaturesNV {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          shadingRateImage;
    VkBool32          shadingRateCoarseSampleOrder;
} VkPhysicalDeviceShadingRateImageFeaturesNV;
```

The members of the **VkPhysicalDeviceShadingRateImageFeaturesNV** structure describe the following features:

- **shadingRateImage** indicates that the implementation supports the use of a shading rate image to derive an effective shading rate for fragment processing. It also indicates that the implementation supports the **ShadingRateNV** SPIR-V execution mode.
- **shadingRateCoarseSampleOrder** indicates that the implementation supports a user-configurable ordering of coverage samples in fragments larger than one pixel.

See [Shading Rate Image](#) for more information.

If the **VkPhysicalDeviceShadingRateImageFeaturesNV** structure is included in the **pNext** chain of **VkPhysicalDeviceFeatures2**, it is filled with values indicating whether the feature is supported. **VkPhysicalDeviceShadingRateImageFeaturesNV** **can** also be used in the **pNext** chain of **VkDeviceCreateInfo** to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_FEATURES_NV`

The `VkPhysicalDeviceFragmentDensityMapFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceFragmentDensityMapFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            fragmentDensityMap;
    VkBool32            fragmentDensityMapDynamic;
    VkBool32            fragmentDensityMapNonSubsampledImages;
} VkPhysicalDeviceFragmentDensityMapFeaturesEXT;
```

The members of the `VkPhysicalDeviceFragmentDensityMapFeaturesEXT` structure describe the following features:

- `fragmentDensityMap` specifies whether the implementation supports render passes with a fragment density map attachment. If this feature is not enabled and the `pNext` chain of `VkRenderPassCreateInfo` contains `VkRenderPassFragmentDensityMapCreateInfoEXT`, `fragmentDensityMapAttachment` must be `VK_ATTACHMENT_UNUSED`.
- `fragmentDensityMapDynamic` specifies whether the implementation supports dynamic fragment density map image views. If this feature is not enabled, `VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT` must not be included in `VkImageViewCreateInfo::flags`.
- `fragmentDensityMapNonSubsampledImages` specifies whether the implementation supports regular non-subsampled image attachments with fragment density map render passes. If this feature is not enabled, render passes with a `fragment density map attachment` must only have `subsampled attachments` bound.

If the `VkPhysicalDeviceFragmentDensityMapFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceFragmentDensityMapFeaturesEXT` can also be used in `pNext` chain of `VkDeviceCreateInfo` to enable the features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_FEATURES_EXT`

The `VkPhysicalDeviceScalarBlockLayoutFeaturesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceScalarBlockLayoutFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           scalarBlockLayout;
} VkPhysicalDeviceScalarBlockLayoutFeaturesEXT;

```

The members of the `VkPhysicalDeviceScalarBlockLayoutFeaturesEXT` structure describe the following features:

- `scalarBlockLayout` indicates that the implementation supports the layout of resource blocks in shaders using [scalar alignment](#).

If the `VkPhysicalDeviceScalarBlockLayoutFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceScalarBlockLayoutFeaturesEXT` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable this feature.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES_EXT`

The `VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR` structure is defined as:

```

typedef struct VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           uniformBufferStandardLayout;
} VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR;

```

The members of the `VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR` structure describe the following features:

- `uniformBufferStandardLayout` indicates that the implementation supports the same layouts for uniform buffers as for storage and other kinds of buffers. See [Standard Buffer Layout](#).

If the `VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable this feature.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES_KHR`

The `VkPhysicalDeviceDepthClipEnableFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceDepthClipEnableFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          depthClipEnable;
} VkPhysicalDeviceDepthClipEnableFeaturesEXT;
```

The members of the `VkPhysicalDeviceDepthClipEnableFeaturesEXT` structure describe the following features:

- `depthClipEnable` indicates that the implementation supports setting the depth clipping operation explicitly via the `VkPipelineRasterizationDepthClipStateCreateInfoEXT` pipeline state. Otherwise depth clipping is only enabled when `VkPipelineRasterizationStateCreateInfo::depthClampEnable` is set to `VK_FALSE`.

If the `VkPhysicalDeviceDepthClipEnableFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceDepthClipEnableFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable this feature.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_CLIP_ENABLE_FEATURES_EXT`

The `VkPhysicalDeviceMemoryPriorityFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceMemoryPriorityFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          memoryPriority;
} VkPhysicalDeviceMemoryPriorityFeaturesEXT;
```

The members of the `VkPhysicalDeviceMemoryPriorityFeaturesEXT` structure describe the following features:

- `memoryPriority` indicates that the implementation supports memory priorities specified at memory allocation time via `VkMemoryPriorityAllocateInfoEXT`.

If the `VkPhysicalDeviceMemoryPriorityFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceMemoryPriorityFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PRIORITY_FEATURES_EXT`

The `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceBufferDeviceAddressFeaturesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            bufferDeviceAddress;
    VkBool32            bufferDeviceAddressCaptureReplay;
    VkBool32            bufferDeviceAddressMultiDevice;
} VkPhysicalDeviceBufferDeviceAddressFeaturesKHR;
```

The members of the `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR` structure describe the following features:

- `bufferDeviceAddress` indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from `vkGetBufferDeviceAddressKHR`.
- `bufferDeviceAddressCaptureReplay` indicates that the implementation supports saving and reusing buffer and device addresses, e.g. for trace capture and replay.
- `bufferDeviceAddressMultiDevice` indicates that the implementation supports the `bufferDeviceAddress` feature for logical devices created with multiple physical devices. If this feature is not supported, buffer addresses **must** not be queried on a logical device created with more than one physical device.

*Note*



`bufferDeviceAddressMultiDevice` exists to allow certain legacy platforms to be able to support `bufferDeviceAddress` without needing to support shared GPU virtual addresses for multi-device configurations.

See `vkGetBufferDeviceAddressKHR` for more information.

If the `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_KHR`

The `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceBufferDeviceAddressFeaturesEXT {
    VkStructureType    sType;
    void*            pNext;
    VkBool32          bufferDeviceAddress;
    VkBool32          bufferDeviceAddressCaptureReplay;
    VkBool32          bufferDeviceAddressMultiDevice;
} VkPhysicalDeviceBufferDeviceAddressFeaturesEXT;

```

The members of the `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` structure describe the following features:

- `bufferDeviceAddress` indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from `vkGetBufferDeviceAddressEXT`.
- `bufferDeviceAddressCaptureReplay` indicates that the implementation supports saving and reusing buffer addresses, e.g. for trace capture and replay.
- `bufferDeviceAddressMultiDevice` indicates that the implementation supports the `bufferDeviceAddress` feature for logical devices created with multiple physical devices. If this feature is not supported, buffer addresses **must** not be queried on a logical device created with more than one physical device.

If the `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

*Note*

The `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` structure has the same members as the `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR` structure, but the functionality indicated by the members is expressed differently. The features indicated by the `VkPhysicalDeviceBufferDeviceAddressFeaturesKHR` structure requires additional flags to be passed at memory allocation time, and the capture and replay mechanism is built around opaque capture addresses for buffer and memory objects.



## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT`

The `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV` structure is defined as:

```

typedef struct VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV {
    VkStructureType      sType;
    void*                pNext;
    VkBool32              dedicatedAllocationImageAliasing;
} VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV;

```

The members of the `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV` structure describe the following features:

- `dedicatedAllocationImageAliasing` indicates that the implementation supports aliasing of compatible image objects on a dedicated allocation.

If the `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- **sType must be** `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEDICATED_ALLOCATION_IMAGE_ALIASING_FEATURES_NV`

The `VkPhysicalDeviceImagelessFramebufferFeaturesKHR` structure is defined as:

```

typedef struct VkPhysicalDeviceImagelessFramebufferFeaturesKHR {
    VkStructureType      sType;
    void*                pNext;
    VkBool32              imagelessFramebuffer;
} VkPhysicalDeviceImagelessFramebufferFeaturesKHR;

```

The members of the `VkPhysicalDeviceImagelessFramebufferFeaturesKHR` structure describe the following features:

- `imagelessFramebuffer` indicates that the implementation supports specifying the image view for attachments at render pass begin time via `VkRenderPassAttachmentBeginInfoKHR`.

If the `VkPhysicalDeviceImagelessFramebufferFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceImagelessFramebufferFeaturesKHR` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable this feature.

### Valid Usage (Implicit)

- **sType must be** `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES_KHR`

The `VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT {
    VkStructureType    sType;
    void*            pNext;
    VkBool32           fragmentShaderSampleInterlock;
    VkBool32           fragmentShaderPixelInterlock;
    VkBool32           fragmentShaderShadingRateInterlock;
} VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT;

```

The members of the `VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT` structure describe the following features:

- `fragmentShaderSampleInterlock` indicates that the implementation supports the `FragmentShaderSampleInterlockEXT` SPIR-V capability.
- `fragmentShaderPixelInterlock` indicates that the implementation supports the `FragmentShaderPixelInterlockEXT` SPIR-V capability.
- `fragmentShaderShadingRateInterlock` indicates that the implementation supports the `FragmentShaderShadingRateInterlockEXT` SPIR-V capability.

If the `VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT`

The `VkPhysicalDeviceCooperativeMatrixFeaturesNV` structure is defined as:

```

typedef struct VkPhysicalDeviceCooperativeMatrixFeaturesNV {
    VkStructureType    sType;
    void*            pNext;
    VkBool32           cooperativeMatrix;
    VkBool32           cooperativeMatrixRobustBufferAccess;
} VkPhysicalDeviceCooperativeMatrixFeaturesNV;

```

The members of the `VkPhysicalDeviceCooperativeMatrixFeaturesNV` structure describe the following features:

- `cooperativeMatrix` indicates that the implementation supports the `CooperativeMatrixNV` SPIR-V capability.
- `cooperativeMatrixRobustBufferAccess` indicates that the implementation supports robust buffer access for SPIR-V `OpCooperativeMatrixLoadNV` and `OpCooperativeMatrixStoreNV` instructions.

If the `VkPhysicalDeviceCooperativeMatrixFeaturesNV` structure is included in the `pNext` chain of

`VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceCooperativeMatrixFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_FEATURES_NV`

The `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceYcbcrImageArraysFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          ycbcrImageArrays;
} VkPhysicalDeviceYcbcrImageArraysFeaturesEXT;
```

The members of the `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` structure describe the following features:

- `ycbcrImageArrays` indicates that the implementation supports creating images with a format that requires `YCBCR` conversion and has multiple array layers.

If the `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT`

The `VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR {
    VkStructureType    sType;
    void*             pNext;
    VkBool32          shaderSubgroupExtendedTypes;
} VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR;
```

The members of the `VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR` structure describe the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shaderSubgroupExtendedTypes` is a boolean that specifies whether subgroup operations can use 8-

bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types if the implementation supports the types.

If the `VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES_KHR`

The `VkPhysicalDeviceHostQueryResetFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceHostQueryResetFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            hostQueryReset;
} VkPhysicalDeviceHostQueryResetFeaturesEXT;
```

The members of the `VkPhysicalDeviceHostQueryResetFeaturesEXT` structure describe the following features:

- `hostQueryReset` indicates that the implementation supports resetting queries from the host with `vkResetQueryPoolEXT`.

If the `VkPhysicalDeviceHostQueryResetFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceHostQueryResetFeaturesEXT` **can** also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES_EXT`

The `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            shaderIntegerFunctions2;
} VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL;
```

The members of the `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL` structure describe the following features:

- `shaderIntegerFunctions2` indicates that the implementation supports the `ShaderIntegerFunctions2INTEL` SPIR-V capability.

If the `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_FUNCTIONS_2_FEATURES_INTEL`

The `VkPhysicalDeviceCoverageReductionModeFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceCoverageReductionModeFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            coverageReductionMode;
} VkPhysicalDeviceCoverageReductionModeFeaturesNV;
```

The members of the `VkPhysicalDeviceCoverageReductionModeFeaturesNV` structure describe the following features:

- `coverageReductionMode` indicates whether the implementation supports coverage reduction modes. See [Coverage Reduction](#).

If the `VkPhysicalDeviceCoverageReductionModeFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceCoverageReductionModeFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COVERAGE_REDUCTION_MODE_FEATURES_NV`

The `VkPhysicalDeviceTimelineSemaphoreFeaturesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceTimelineSemaphoreFeaturesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            timelineSemaphore;
} VkPhysicalDeviceTimelineSemaphoreFeaturesKHR;
```

The members of the `VkPhysicalDeviceTimelineSemaphoreFeaturesKHR` structure describe the following features:

- `timelineSemaphore` indicates whether semaphores created with a `VkSemaphoreTypeKHR` of `VK_SEMAPHORE_TYPE_TIMELINE_KHR` are supported.

If the `VkPhysicalDeviceTimelineSemaphoreFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceTimelineSemaphoreFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES_KHR`

The `VkPhysicalDeviceIndexTypeUInt8FeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceIndexTypeUInt8FeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            indexTypeUInt8;
} VkPhysicalDeviceIndexTypeUInt8FeaturesEXT;
```

The members of the `VkPhysicalDeviceIndexTypeUInt8FeaturesEXT` structure describe the following features:

- `indexTypeUInt8` indicates that `VK_INDEX_TYPE_UINT8_EXT` can be used with `vkCmdBindIndexBuffer`.

If the `VkPhysicalDeviceIndexTypeUInt8FeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceIndexTypeUInt8FeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT`

The `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderSMBuiltinsFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            shaderSMBuiltins;
} VkPhysicalDeviceShaderSMBuiltinsFeaturesNV;
```

The members of the `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV` structure describe the following features:

- `shaderSMBuiltins` indicates whether the implementation supports the SPIR-V `ShaderSMBuiltinsNV`

capability.

If the `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_FEATURES_NV`

The `VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR {  
    VkStructureType    sType;  
    void*             pNext;  
    VkBool32          separateDepthStencilLayouts;  
} VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR;
```

The members of the `VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR` structure describe the following features:

- `separateDepthStencilLayouts` indicates whether the implementation supports a `VkImageMemoryBarrier` for a depth/stencil image with only one of `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` set, and whether `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR` can be used.

If the `VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES_KHR`

The `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR` structure is defined as:

```
typedef struct VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR {  
    VkStructureType    sType;  
    void*             pNext;  
    VkBool32          pipelineExecutableInfo;  
} VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR;
```

The members of the `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR` structure describe the following features:

- `pipelineExecutableInfo` indicates that the implementation supports reporting properties and statistics about the executables associated with a compiled pipeline.

If the `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable features.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_EXECUTABLE_PROPERTIES_FEATURES_KHR`

The `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT {  
    VkStructureType    sType;  
    void*             pNext;  
    VkBool32          shaderDemoteToHelperInvocation;  
} VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT;
```

The members of the `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT` structure describe the following features:

- `shaderDemoteToHelperInvocation` indicates whether the implementation supports the SPIR-V `DemoteToHelperInvocationEXT` capability.

If the `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT`

The `VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           texelBufferAlignment;
} VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT;

```

The members of the `VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT` structure describe the following features:

- `texelBufferAlignment` indicates whether the implementation uses more specific alignment requirements advertised in `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT` rather than `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`.

If the `VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT`

The `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           textureCompressionASTC_HDR;
} VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT;

```

The members of the `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` structure describe the following features:

- `textureCompressionASTC_HDR` indicates whether all of the ASTC HDR compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for the following formats:

- `VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT`

- VK\_FORMAT\_ASTC\_8x8\_SFLOAT\_BLOCK\_EXT
- VK\_FORMAT\_ASTC\_10x5\_SFLOAT\_BLOCK\_EXT
- VK\_FORMAT\_ASTC\_10x6\_SFLOAT\_BLOCK\_EXT
- VK\_FORMAT\_ASTC\_10x8\_SFLOAT\_BLOCK\_EXT
- VK\_FORMAT\_ASTC\_10x10\_SFLOAT\_BLOCK\_EXT
- VK\_FORMAT\_ASTC\_12x10\_SFLOAT\_BLOCK\_EXT
- VK\_FORMAT\_ASTC\_12x12\_SFLOAT\_BLOCK\_EXT

To query for additional properties, or if the feature is not enabled, [vkGetPhysicalDeviceFormatProperties](#) and [vkGetPhysicalDeviceImageFormatProperties](#) can be used to check for supported properties of individual formats as normal.

If the `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` can also be used in the `pNext` chain of `vkCreateDevice` to enable features.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT`

The `VkPhysicalDeviceLineRasterizationFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceLineRasterizationFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            rectangularLines;
    VkBool32            bresenhamLines;
    VkBool32            smoothLines;
    VkBool32            stippledRectangularLines;
    VkBool32            stippledBresenhamLines;
    VkBool32            stippledSmoothLines;
} VkPhysicalDeviceLineRasterizationFeaturesEXT;
```

The members of the `VkPhysicalDeviceLineRasterizationFeaturesEXT` structure describe the following features:

- `rectangularLines` indicates whether the implementation supports [rectangular line rasterization](#).
- `bresenhamLines` indicates whether the implementation supports [Bresenham-style line rasterization](#).
- `smoothLines` indicates whether the implementation supports [smooth line rasterization](#).
- `stippledRectangularLines` indicates whether the implementation supports [stippled line rasterization](#) with `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` lines, or with `VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT` lines if `VkPhysicalDeviceLimits::strictLines` is `VK_TRUE`.

- `stippledBresenhamLines` indicates whether the implementation supports stippled line rasterization with `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT` lines.
- `stippledSmoothLines` indicates whether the implementation supports stippled line rasterization with `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` lines.

If the `VkPhysicalDeviceLineRasterizationFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceLineRasterizationFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT`

The `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceSubgroupSizeControlFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            subgroupSizeControl;
    VkBool32            computeFullSubgroups;
} VkPhysicalDeviceSubgroupSizeControlFeaturesEXT;
```

The members of the `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure describe the following features:

- `subgroupSizeControl` indicates whether the implementation supports controlling shader subgroup sizes via the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag and the `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure.
- `computeFullSubgroups` indicates whether the implementation supports requiring full subgroups in compute shaders via the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` flag.

If the `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceFeatures2`, it is filled with values indicating whether the feature is supported. `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to enable the feature.

### Note

The `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure was added in version 2 of the `VK_EXT_subgroup_size_control` extension. Version 1 implementations of this extension will not fill out the features structure but applications may assume that both `subgroupSizeControl` and `computeFullSubgroups` are supported if the extension is supported. (See also the [Feature Requirements](#) section.) Applications are advised to add a `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure to the `pNext` chain of `VkDeviceCreateInfo` to enable the features regardless of the version of the extension supported by the implementation. If the implementation only supports version 1, it will safely ignore the `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` structure.



### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES_EXT`

The `VkPhysicalDeviceCoherentMemoryFeaturesAMD` structure is defined as:

```
typedef struct VkPhysicalDeviceCoherentMemoryFeaturesAMD {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            deviceCoherentMemory;
} VkPhysicalDeviceCoherentMemoryFeaturesAMD;
```

The members of the `VkPhysicalDeviceCoherentMemoryFeaturesAMD` structure describe the following features:

- `deviceCoherentMemory` indicates that the implementation supports [device coherent memory](#).

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COHERENT_MEMORY_FEATURES_AMD`

## 35.1. Feature Requirements

All Vulkan graphics implementations **must** support the following features:

- `robustBufferAccess`
- `multiview`, if Vulkan 1.1 is supported.
- `uniformBufferStandardLayout`, if the `VK_KHR_uniform_buffer_standard_layout` extension is supported.
- `variablePointersStorageBuffer`, if the `VK_KHR_variable_pointers` extension is supported.

- `storageBuffer8BitAccess`, if the `VK_KHR_8bit_storage` extension is supported.
- If the `VK_EXT_descriptor_indexing` extension is supported:
  - `shaderSampledImageArrayDynamicIndexing`
  - `shaderStorageBufferArrayDynamicIndexing`
  - `shaderUniformTexelBufferArrayDynamicIndexing`
  - `shaderStorageTexelBufferArrayDynamicIndexing`
  - `shaderSampledImageArrayNonUniformIndexing`
  - `shaderStorageBufferArrayNonUniformIndexing`
  - `shaderUniformTexelBufferArrayNonUniformIndexing`
  - `descriptorBindingSampledImageUpdateAfterBind`
  - `descriptorBindingStorageImageUpdateAfterBind`
  - `descriptorBindingStorageBufferUpdateAfterBind` (see also `robustBufferAccessUpdateAfterBind`)
  - `descriptorBindingUniformTexelBufferUpdateAfterBind` (see also `robustBufferAccessUpdateAfterBind`)
  - `descriptorBindingStorageTexelBufferUpdateAfterBind` (see also `robustBufferAccessUpdateAfterBind`)
  - `descriptorBindingUpdateUnusedWhilePending`
  - `descriptorBindingPartiallyBound`
  - `runtimeDescriptorArray`
- `inlineUniformBlock`, if the `VK_EXT_inline_uniform_block` extension is supported.
- `descriptorBindingInlineUniformBlockUpdateAfterBind`, if the `VK_EXT_inline_uniform_block` and `VK_EXT_descriptor_indexing` extensions are both supported.
- `scalarBlockLayout`, if the `VK_EXT_scalar_block_layout` extension is supported.
- `subgroupSizeControl`, if the `VK_EXT_subgroup_size_control` extension is supported.
- `computeFullSubgroups`, if the `VK_EXT_subgroup_size_control` extension is supported.
- `timelineSemaphore`, if the `VK_KHR_timeline_semaphore` extension is supported.

All other features defined in the Specification are **optional**.

# Chapter 36. Limits

*Limits* are implementation-dependent minimums, maximums, and other device characteristics that an application **may** need to be aware of.

## Note

Limits are reported via the basic `VkPhysicalDeviceLimits` structure, as well as the extensible structure `VkPhysicalDeviceProperties2`, which was added in `VK_KHR_get_physical_device_properties2` and included in Vulkan 1.1. When limits are added in future Vulkan versions or extensions, each extension **should** introduce one new limit structure, if needed. This structures **can** be added to the `pNext` chain of the `VkPhysicalDeviceProperties2` structure.



The `VkPhysicalDeviceLimits` structure is defined as:

```
typedef struct VkPhysicalDeviceLimits {  
    uint32_t          maxImageDimension1D;  
    uint32_t          maxImageDimension2D;  
    uint32_t          maxImageDimension3D;  
    uint32_t          maxImageDimensionCube;  
    uint32_t          maxImageArrayLayers;  
    uint32_t          maxTexelBufferElements;  
    uint32_t          maxUniformBufferRange;  
    uint32_t          maxStorageBufferRange;  
    uint32_t          maxPushConstantsSize;  
    uint32_t          maxMemoryAllocationCount;  
    uint32_t          maxSamplerAllocationCount;  
    VkDeviceSize      bufferImageGranularity;  
    VkDeviceSize      sparseAddressSpaceSize;  
    uint32_t          maxBoundDescriptorSets;  
    uint32_t          maxPerStageDescriptorSamplers;  
    uint32_t          maxPerStageDescriptorUniformBuffers;  
    uint32_t          maxPerStageDescriptorStorageBuffers;  
    uint32_t          maxPerStageDescriptorSampledImages;  
    uint32_t          maxPerStageDescriptorStorageImages;  
    uint32_t          maxPerStageDescriptorInputAttachments;  
    uint32_t          maxPerStageResources;  
    uint32_t          maxDescriptorSetSamplers;  
    uint32_t          maxDescriptorSetUniformBuffers;  
    uint32_t          maxDescriptorSetUniformBuffersDynamic;  
    uint32_t          maxDescriptorSetStorageBuffers;  
    uint32_t          maxDescriptorSetStorageBuffersDynamic;  
    uint32_t          maxDescriptorSetSampledImages;  
    uint32_t          maxDescriptorSetStorageImages;  
    uint32_t          maxDescriptorSetInputAttachments;  
    uint32_t          maxVertexInputAttributes;  
    uint32_t          maxVertexInputBindings;  
    uint32_t          maxVertexInputAttributeOffset;  
    uint32_t          maxVertexInputBindingStride;
```

```
uint32_t          maxVertexOutputComponents;
uint32_t          maxTessellationGenerationLevel;
uint32_t          maxTessellationPatchSize;
uint32_t          maxTessellationControlPerVertexInputComponents;
uint32_t          maxTessellationControlPerVertexOutputComponents;
uint32_t          maxTessellationControlPerPatchOutputComponents;
uint32_t          maxTessellationControlTotalOutputComponents;
uint32_t          maxTessellationEvaluationInputComponents;
uint32_t          maxTessellationEvaluationOutputComponents;
uint32_t          maxGeometryShaderInvocations;
uint32_t          maxGeometryInputComponents;
uint32_t          maxGeometryOutputComponents;
uint32_t          maxGeometryOutputVertices;
uint32_t          maxGeometryTotalOutputComponents;
uint32_t          maxFragmentInputComponents;
uint32_t          maxFragmentOutputAttachments;
uint32_t          maxFragmentDualSrcAttachments;
uint32_t          maxFragmentCombinedOutputResources;
uint32_t          maxComputeSharedMemorySize;
uint32_t          maxComputeWorkGroupCount[3];
uint32_t          maxComputeWorkGroupInvocations;
uint32_t          maxComputeWorkGroupSize[3];
uint32_t          subPixelPrecisionBits;
uint32_t          subTexelPrecisionBits;
uint32_t          mipmapPrecisionBits;
uint32_t          maxDrawIndexedIndexValue;
uint32_t          maxDrawIndirectCount;
float             maxSamplerLodBias;
float             maxSamplerAnisotropy;
uint32_t          maxViewports;
uint32_t          maxViewportDimensions[2];
float             viewportBoundsRange[2];
uint32_t          viewportSubPixelBits;
size_t            minMemoryMapAlignment;
VkDeviceSize      minTexelBufferOffsetAlignment;
VkDeviceSize      minUniformBufferOffsetAlignment;
VkDeviceSize      minStorageBufferOffsetAlignment;
int32_t           minTexelOffset;
uint32_t          maxTexelOffset;
int32_t           minTexelGatherOffset;
uint32_t          maxTexelGatherOffset;
float             minInterpolationOffset;
float             maxInterpolationOffset;
uint32_t          subPixelInterpolationOffsetBits;
uint32_t          maxFramebufferWidth;
uint32_t          maxFramebufferHeight;
uint32_t          maxFramebufferLayers;
VkSampleCountFlags framebufferColorSampleCounts;
VkSampleCountFlags framebufferDepthSampleCounts;
VkSampleCountFlags framebufferStencilSampleCounts;
VkSampleCountFlags framebufferNoAttachmentsSampleCounts;
```

```

        uint32_t          maxColorAttachments;
        VkSampleCountFlags sampledImageColorSampleCounts;
        VkSampleCountFlags sampledImageIntegerSampleCounts;
        VkSampleCountFlags sampledImageDepthSampleCounts;
        VkSampleCountFlags sampledImageStencilSampleCounts;
        VkSampleCountFlags storageImageSampleCounts;
        uint32_t          maxSampleMaskWords;
        VkBool32           timestampComputeAndGraphics;
        float              timestampPeriod;
        uint32_t          maxClipDistances;
        uint32_t          maxCullDistances;
        uint32_t          maxCombinedClipAndCullDistances;
        uint32_t          discreteQueuePriorities;
        float              pointSizeRange[2];
        float              lineWidthRange[2];
        float              pointSizeGranularity;
        float              lineWidthGranularity;
        VkBool32           strictLines;
        VkBool32           standardSampleLocations;
        VkDeviceSize        optimalBufferCopyOffsetAlignment;
        VkDeviceSize        optimalBufferCopyRowPitchAlignment;
        VkDeviceSize        nonCoherentAtomSize;
    } VkPhysicalDeviceLimits;
}

```

The `VkPhysicalDeviceLimits` are properties of the physical device. These are available in the `limits` member of the `VkPhysicalDeviceProperties` structure which is returned from `vkGetPhysicalDeviceProperties`.

- `maxImageDimension1D` is the maximum dimension (`width`) supported for all images created with an `imageType` of `VK_IMAGE_TYPE_1D`.
- `maxImageDimension2D` is the maximum dimension (`width` or `height`) supported for all images created with an `imageType` of `VK_IMAGE_TYPE_2D` and without `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` set in `flags`.
- `maxImageDimension3D` is the maximum dimension (`width`, `height`, or `depth`) supported for all images created with an `imageType` of `VK_IMAGE_TYPE_3D`.
- `maxImageDimensionCube` is the maximum dimension (`width` or `height`) supported for all images created with an `imageType` of `VK_IMAGE_TYPE_2D` and with `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` set in `flags`.
- `maxImageArrayLayers` is the maximum number of layers (`arrayLayers`) for an image.
- `maxTexelBufferElements` is the maximum number of addressable texels for a buffer view created on a buffer which was created with the `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` set in the `usage` member of the `VkBufferCreateInfo` structure.
- `maxUniformBufferRange` is the maximum value that `can` be specified in the `range` member of any `VkDescriptorBufferInfo` structures passed to a call to `vkUpdateDescriptorSets` for descriptors of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`.

- `maxStorageBufferRange` is the maximum value that **can** be specified in the `range` member of any `VkDescriptorBufferInfo` structures passed to a call to `vkUpdateDescriptorSets` for descriptors of type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`.
- `maxPushConstantsSize` is the maximum size, in bytes, of the pool of push constant memory. For each of the push constant ranges indicated by the `pPushConstantRanges` member of the `VkPipelineLayoutCreateInfo` structure, `(offset + size)` **must** be less than or equal to this limit.
- `maxMemoryAllocationCount` is the maximum number of device memory allocations, as created by `vkAllocateMemory`, which **can** simultaneously exist.
- `maxSamplerAllocationCount` is the maximum number of sampler objects, as created by `vkCreateSampler`, which **can** simultaneously exist on a device.
- `bufferImageGranularity` is the granularity, in bytes, at which buffer or linear image resources, and optimal image resources **can** be bound to adjacent offsets in the same `VkDeviceMemory` object without aliasing. See [Buffer-Image Granularity](#) for more details.
- `sparseAddressSpaceSize` is the total amount of address space available, in bytes, for sparse memory resources. This is an upper bound on the sum of the size of all sparse resources, regardless of whether any memory is bound to them.
- `maxBoundDescriptorSets` is the maximum number of descriptor sets that **can** be simultaneously used by a pipeline. All `DescriptorSet` decorations in shader modules **must** have a value less than `maxBoundDescriptorSets`. See [Descriptor Sets](#).
- `maxPerStageDescriptorSamplers` is the maximum number of samplers that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. A descriptor is accessible to a shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See [Sampler](#) and [Combined Image Sampler](#).
- `maxPerStageDescriptorUniformBuffers` is the maximum number of uniform buffers that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. A descriptor is accessible to a shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See [Uniform Buffer](#) and [Dynamic Uniform Buffer](#).
- `maxPerStageDescriptorStorageBuffers` is the maximum number of storage buffers that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See [Storage Buffer](#) and [Dynamic Storage Buffer](#).
- `maxPerStageDescriptorSampledImages` is the maximum number of sampled images that **can** be

accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, or `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See [Combined Image Sampler](#), [Sampled Image](#), and [Uniform Texel Buffer](#).

- `maxPerStageDescriptorStorageImages` is the maximum number of storage images that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See [Storage Image](#), and [Storage Texel Buffer](#).
- `maxPerStageDescriptorInputAttachments` is the maximum number of input attachments that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. These are only supported for the fragment stage. See [Input Attachment](#).
- `maxPerStageResources` is the maximum number of resources that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. For the fragment shader stage the framebuffer color attachments also count against this limit.
- `maxDescriptorSetSamplers` is the maximum number of samplers that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Sampler](#) and [Combined Image Sampler](#).
- `maxDescriptorSetUniformBuffers` is the maximum number of uniform buffers that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Uniform Buffer](#) and [Dynamic Uniform Buffer](#).

- `maxDescriptorSetUniformBuffersDynamic` is the maximum number of dynamic uniform buffers that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Dynamic Uniform Buffer](#).
- `maxDescriptorSetStorageBuffers` is the maximum number of storage buffers that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Storage Buffer](#) and [Dynamic Storage Buffer](#).
- `maxDescriptorSetStorageBuffersDynamic` is the maximum number of dynamic storage buffers that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Dynamic Storage Buffer](#).
- `maxDescriptorSetSampledImages` is the maximum number of sampled images that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, or `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Combined Image Sampler](#), [Sampled Image](#), and [Uniform Texel Buffer](#).
- `maxDescriptorSetStorageImages` is the maximum number of storage images that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Storage Image](#), and [Storage Texel Buffer](#).
- `maxDescriptorSetInputAttachments` is the maximum number of input attachments that **can** be included in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit. See [Input Attachment](#).
- `maxVertexInputAttributes` is the maximum number of vertex input attributes that **can** be specified for a graphics pipeline. These are described in the array of `VkVertexInputAttributeDescription` structures that are provided at graphics pipeline creation time via the `pVertexAttributeDescriptions` member of the `VkPipelineVertexInputStateCreateInfo` structure. See [Vertex Attributes](#) and [Vertex Input Description](#).
- `maxVertexInputBindings` is the maximum number of vertex buffers that **can** be specified for providing vertex attributes to a graphics pipeline. These are described in the array of `VkVertexInputBindingDescription` structures that are provided at graphics pipeline creation time via the `pVertexBindingDescriptions` member of the `VkPipelineVertexInputStateCreateInfo` structure. The `binding` member of `VkVertexInputBindingDescription` **must** be less than this limit.

See [Vertex Input Description](#).

- `maxVertexInputAttributeOffset` is the maximum vertex input attribute offset that **can** be added to the vertex input binding stride. The `offset` member of the `VkVertexInputAttributeDescription` structure **must** be less than or equal to this limit. See [Vertex Input Description](#).
- `maxVertexInputBindingStride` is the maximum vertex input binding stride that **can** be specified in a vertex input binding. The `stride` member of the `VkVertexInputBindingDescription` structure **must** be less than or equal to this limit. See [Vertex Input Description](#).
- `maxVertexOutputComponents` is the maximum number of components of output variables which **can** be output by a vertex shader. See [Vertex Shaders](#).
- `maxTessellationGenerationLevel` is the maximum tessellation generation level supported by the fixed-function tessellation primitive generator. See [Tessellation](#).
- `maxTessellationPatchSize` is the maximum patch size, in vertices, of patches that **can** be processed by the tessellation control shader and tessellation primitive generator. The `patchControlPoints` member of the `VkPipelineTessellationStateCreateInfo` structure specified at pipeline creation time and the value provided in the `OutputVertices` execution mode of shader modules **must** be less than or equal to this limit. See [Tessellation](#).
- `maxTessellationControlPerVertexInputComponents` is the maximum number of components of input variables which **can** be provided as per-vertex inputs to the tessellation control shader stage.
- `maxTessellationControlPerVertexOutputComponents` is the maximum number of components of per-vertex output variables which **can** be output from the tessellation control shader stage.
- `maxTessellationControlPerPatchOutputComponents` is the maximum number of components of per-patch output variables which **can** be output from the tessellation control shader stage.
- `maxTessellationControlTotalOutputComponents` is the maximum total number of components of per-vertex and per-patch output variables which **can** be output from the tessellation control shader stage.
- `maxTessellationEvaluationInputComponents` is the maximum number of components of input variables which **can** be provided as per-vertex inputs to the tessellation evaluation shader stage.
- `maxTessellationEvaluationOutputComponents` is the maximum number of components of per-vertex output variables which **can** be output from the tessellation evaluation shader stage.
- `maxGeometryShaderInvocations` is the maximum invocation count supported for instanced geometry shaders. The value provided in the `Invocations` execution mode of shader modules **must** be less than or equal to this limit. See [Geometry Shading](#).
- `maxGeometryInputComponents` is the maximum number of components of input variables which **can** be provided as inputs to the geometry shader stage.
- `maxGeometryOutputComponents` is the maximum number of components of output variables which **can** be output from the geometry shader stage.
- `maxGeometryOutputVertices` is the maximum number of vertices which **can** be emitted by any geometry shader.
- `maxGeometryTotalOutputComponents` is the maximum total number of components of output, across all emitted vertices, which **can** be output from the geometry shader stage.

- `maxFragmentInputComponents` is the maximum number of components of input variables which **can** be provided as inputs to the fragment shader stage.
- `maxFragmentOutputAttachments` is the maximum number of output attachments which **can** be written to by the fragment shader stage.
- `maxFragmentDualSrcAttachments` is the maximum number of output attachments which **can** be written to by the fragment shader stage when blending is enabled and one of the dual source blend modes is in use. See [Dual-Source Blending](#) and [dualSrcBlend](#).
- `maxFragmentCombinedOutputResources` is the total number of storage buffers, storage images, and output buffers which **can** be used in the fragment shader stage.
- `maxComputeSharedMemorySize` is the maximum total storage size, in bytes, available for variables declared with the `Workgroup` storage class in shader modules (or with the `shared` storage qualifier in GLSL) in the compute shader stage. The amount of storage consumed by the variables declared with the `Workgroup` storage class is implementation-dependent. However, the amount of storage consumed may not exceed the largest block size that would be obtained if all active variables declared with `Workgroup` storage class were assigned offsets in an arbitrary order by successively taking the smallest valid offset according to the [Standard Storage Buffer Layout](#) rules. (This is equivalent to using the GLSL `std430` layout rules.)
- `maxComputeWorkGroupCount`[3] is the maximum number of local workgroups that **can** be dispatched by a single dispatch command. These three values represent the maximum number of local workgroups for the X, Y, and Z dimensions, respectively. The workgroup count parameters to the dispatch commands **must** be less than or equal to the corresponding limit. See [Dispatching Commands](#).
- `maxComputeWorkGroupInvocations` is the maximum total number of compute shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the `LocalSize` execution mode in shader modules or by the object decorated by the `WorkgroupSize` decoration, **must** be less than or equal to this limit.
- `maxComputeWorkGroupSize`[3] is the maximum size of a local compute workgroup, per dimension. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The `x`, `y`, and `z` sizes, as specified by the `LocalSize` execution mode or by the object decorated by the `WorkgroupSize` decoration in shader modules, **must** be less than or equal to the corresponding limit.
- `subPixelPrecisionBits` is the number of bits of subpixel precision in framebuffer coordinates  $x_f$  and  $y_f$ . See [Rasterization](#).
- `subTexelPrecisionBits` is the number of bits of precision in the division along an axis of an image used for minification and magnification filters.  $2^{\text{subTexelPrecisionBits}}$  is the actual number of divisions along each axis of the image represented. Sub-texel values calculated during image sampling will snap to these locations when generating the filtered results.
- `mipmapPrecisionBits` is the number of bits of division that the LOD calculation for mipmap fetching get snapped to when determining the contribution from each mip level to the mip filtered results.  $2^{\text{mipmapPrecisionBits}}$  is the actual number of divisions.
- `maxDrawIndexedIndexValue` is the maximum index value that **can** be used for indexed draw calls when using 32-bit indices. This excludes the primitive restart index value of `0xFFFFFFFF`. See [fullDrawIndexUInt32](#).

- `maxDrawIndirectCount` is the maximum draw count that is supported for indirect draw calls. See [multiDrawIndirect](#).
- `maxSamplerLodBias` is the maximum absolute sampler LOD bias. The sum of the `mipLodBias` member of the `VkSamplerCreateInfo` structure and the `Bias` operand of image sampling operations in shader modules (or 0 if no `Bias` operand is provided to an image sampling operation) are clamped to the range  $[-\text{maxSamplerLodBias}, +\text{maxSamplerLodBias}]$ . See [\[samplers-mipLodBias\]](#).
- `maxSamplerAnisotropy` is the maximum degree of sampler anisotropy. The maximum degree of anisotropic filtering used for an image sampling operation is the minimum of the `maxAnisotropy` member of the `VkSamplerCreateInfo` structure and this limit. See [\[samplers-maxAnisotropy\]](#).
- `maxViewports` is the maximum number of active viewports. The `viewportCount` member of the `VkPipelineViewportStateCreateInfo` structure that is provided at pipeline creation **must** be less than or equal to this limit.
- `maxViewportDimensions[2]` are the maximum viewport dimensions in the X (width) and Y (height) dimensions, respectively. The maximum viewport dimensions **must** be greater than or equal to the largest image which **can** be created and used as a framebuffer attachment. See [Controlling the Viewport](#).
- `viewportBoundsRange[2]` is the [minimum, maximum] range that the corners of a viewport **must** be contained in. This range **must** be at least  $[-2 \times \text{size}, 2 \times \text{size} - 1]$ , where `size` =  $\max(\text{maxViewportDimensions}[0], \text{maxViewportDimensions}[1])$ . See [Controlling the Viewport](#).

*Note*



The intent of the `viewportBoundsRange` limit is to allow a maximum sized viewport to be arbitrarily shifted relative to the output target as long as at least some portion intersects. This would give a bounds limit of  $[-\text{size} + 1, 2 \times \text{size} - 1]$  which would allow all possible non-empty-set intersections of the output target and the viewport. Since these numbers are typically powers of two, picking the signed number range using the smallest possible number of bits ends up with the specified range.

- `viewportSubPixelBits` is the number of bits of subpixel precision for viewport bounds. The subpixel precision that floating-point viewport bounds are interpreted at is given by this limit.
- `minMemoryMapAlignment` is the minimum **required** alignment, in bytes, of host visible memory allocations within the host address space. When mapping a memory allocation with `vkMapMemory`, subtracting `offset` bytes from the returned pointer will always produce an integer multiple of this limit. See [Host Access to Device Memory Objects](#).
- `minTexelBufferOffsetAlignment` is the minimum **required** alignment, in bytes, for the `offset` member of the `VkBufferViewCreateInfo` structure for texel buffers. If `texelBufferAlignment` is enabled, this limit is equivalent to the maximum of the `uniformTexelBufferOffsetAlignmentBytes` and `storageTexelBufferOffsetAlignmentBytes` members of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT`, but smaller alignment is optionally allowed by `storageTexelBufferOffsetSingleTexelAlignment` and `uniformTexelBufferOffsetSingleTexelAlignment`. If `texelBufferAlignment` is not enabled, `VkBufferViewCreateInfo::offset` **must** be a multiple of this value.

- `minUniformBufferOffsetAlignment` is the minimum **required** alignment, in bytes, for the `offset` member of the `VkDescriptorBufferInfo` structure for uniform buffers. When a descriptor of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` is updated, the `offset` **must** be an integer multiple of this limit. Similarly, dynamic offsets for uniform buffers **must** be multiples of this limit.
- `minStorageBufferOffsetAlignment` is the minimum **required** alignment, in bytes, for the `offset` member of the `VkDescriptorBufferInfo` structure for storage buffers. When a descriptor of type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` is updated, the `offset` **must** be an integer multiple of this limit. Similarly, dynamic offsets for storage buffers **must** be multiples of this limit.
- `minTexelOffset` is the minimum offset value for the `ConstOffset` image operand of any of the `OpImageSample*` or `OpImageFetch*` image instructions.
- `maxTexelOffset` is the maximum offset value for the `ConstOffset` image operand of any of the `OpImageSample*` or `OpImageFetch*` image instructions.
- `minTexelGatherOffset` is the minimum offset value for the `Offset`, `ConstOffset`, or `ConstOffsets` image operands of any of the `OpImage*Gather` image instructions.
- `maxTexelGatherOffset` is the maximum offset value for the `Offset`, `ConstOffset`, or `ConstOffsets` image operands of any of the `OpImage*Gather` image instructions.
- `minInterpolationOffset` is the minimum negative offset value for the `offset` operand of the `InterpolateAtOffset` extended instruction.
- `maxInterpolationOffset` is the maximum positive offset value for the `offset` operand of the `InterpolateAtOffset` extended instruction.
- `subPixelInterpolationOffsetBits` is the number of subpixel fractional bits that the `x` and `y` offsets to the `InterpolateAtOffset` extended instruction **may** be rounded to as fixed-point values.
- `maxFramebufferWidth` is the maximum width for a framebuffer. The `width` member of the `VkFramebufferCreateInfo` structure **must** be less than or equal to this limit.
- `maxFramebufferHeight` is the maximum height for a framebuffer. The `height` member of the `VkFramebufferCreateInfo` structure **must** be less than or equal to this limit.
- `maxFramebufferLayers` is the maximum layer count for a layered framebuffer. The `layers` member of the `VkFramebufferCreateInfo` structure **must** be less than or equal to this limit.
- `framebufferColorSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the color sample counts that are supported for all framebuffer color attachments with floating- or fixed-point formats. There is no limit that specifies the color sample counts that are supported for all color attachments with integer formats.
- `framebufferDepthSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the supported depth sample counts for all framebuffer depth/stencil attachments, when the format includes a depth component.
- `framebufferStencilSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the supported stencil sample counts for all framebuffer depth/stencil attachments, when the format includes a stencil component.
- `framebufferNoAttachmentsSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the supported sample counts for a framebuffer with no attachments.

- `maxColorAttachments` is the maximum number of color attachments that **can** be used by a subpass in a render pass. The `colorAttachmentCount` member of the `VkSubpassDescription` structure **must** be less than or equal to this limit.
- `sampledImageColorSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the sample counts supported for all 2D images created with `VK_IMAGE_TILING_OPTIMAL`, `usage` containing `VK_IMAGE_USAGE_SAMPLED_BIT`, and a non-integer color format.
- `sampledImageIntegerSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the sample counts supported for all 2D images created with `VK_IMAGE_TILING_OPTIMAL`, `usage` containing `VK_IMAGE_USAGE_SAMPLED_BIT`, and an integer color format.
- `sampledImageDepthSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the sample counts supported for all 2D images created with `VK_IMAGE_TILING_OPTIMAL`, `usage` containing `VK_IMAGE_USAGE_SAMPLED_BIT`, and a depth format.
- `sampledImageStencilSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the sample supported for all 2D images created with `VK_IMAGE_TILING_OPTIMAL`, `usage` containing `VK_IMAGE_USAGE_SAMPLED_BIT`, and a stencil format.
- `storageImageSampleCounts` is a bitmask<sup>1</sup> of `VkSampleCountFlagBits` indicating the sample counts supported for all 2D images created with `VK_IMAGE_TILING_OPTIMAL`, and `usage` containing `VK_IMAGE_USAGE_STORAGE_BIT`.
- `maxSampleMaskWords` is the maximum number of array elements of a variable decorated with the `SampleMask` built-in decoration.
- `timestampComputeAndGraphics` specifies support for timestamps on all graphics and compute queues. If this limit is set to `VK_TRUE`, all queues that advertise the `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT` in the `VkQueueFamilyProperties::queueFlags` support `VkQueueFamilyProperties::timestampValidBits` of at least 36. See [Timestamp Queries](#).
- `timestampPeriod` is the number of nanoseconds **required** for a timestamp query to be incremented by 1. See [Timestamp Queries](#).
- `maxClipDistances` is the maximum number of clip distances that **can** be used in a single shader stage. The size of any array declared with the `ClipDistance` built-in decoration in a shader module **must** be less than or equal to this limit.
- `maxCullDistances` is the maximum number of cull distances that **can** be used in a single shader stage. The size of any array declared with the `CullDistance` built-in decoration in a shader module **must** be less than or equal to this limit.
- `maxCombinedClipAndCullDistances` is the maximum combined number of clip and cull distances that **can** be used in a single shader stage. The sum of the sizes of any pair of arrays declared with the `ClipDistance` and `CullDistance` built-in decoration used by a single shader stage in a shader module **must** be less than or equal to this limit.
- `discreteQueuePriorities` is the number of discrete priorities that **can** be assigned to a queue based on the value of each member of `VkDeviceQueueCreateInfo::pQueuePriorities`. This **must** be at least 2, and levels **must** be spread evenly over the range, with at least one level at 1.0, and another at 0.0. See [Queue Priority](#).
- `pointSizeRange[2]` is the range [`minimum`,`maximum`] of supported sizes for points. Values written to variables decorated with the `PointSize` built-in decoration are clamped to this range.

- `lineWidthRange[2]` is the range [`minimum`,`maximum`] of supported widths for lines. Values specified by the `lineWidth` member of the `VkPipelineRasterizationStateCreateInfo` or the `lineWidth` parameter to `vkCmdSetLineWidth` are clamped to this range.
- `pointSizeGranularity` is the granularity of supported point sizes. Not all point sizes in the range defined by `pointSizeRange` are supported. This limit specifies the granularity (or increment) between successive supported point sizes.
- `lineWidthGranularity` is the granularity of supported line widths. Not all line widths in the range defined by `lineWidthRange` are supported. This limit specifies the granularity (or increment) between successive supported line widths.
- `strictLines` specifies whether lines are rasterized according to the preferred method of rasterization. If set to `VK_FALSE`, lines **may** be rasterized under a relaxed set of rules. If set to `VK_TRUE`, lines are rasterized as per the strict definition. See [Basic Line Segment Rasterization](#).
- `standardSampleLocations` specifies whether rasterization uses the standard sample locations as documented in [Multisampling](#). If set to `VK_TRUE`, the implementation uses the documented sample locations. If set to `VK_FALSE`, the implementation **may** use different sample locations.
- `optimalBufferCopyOffsetAlignment` is the optimal buffer offset alignment in bytes for `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`. The per texel alignment requirements are enforced, but applications **should** use the optimal alignment for optimal performance and power use.
- `optimalBufferCopyRowPitchAlignment` is the optimal buffer row pitch alignment in bytes for `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`. Row pitch is the number of bytes between texels with the same X coordinate in adjacent rows (Y coordinates differ by one). The per texel alignment requirements are enforced, but applications **should** use the optimal alignment for optimal performance and power use.
- `nonCoherentAtomSize` is the size and alignment in bytes that bounds concurrent access to [host-mapped device memory](#).

1

For all bitmasks of `VkSampleCountFlagBits`, the sample count limits defined above represent the minimum supported sample counts for each image type. Individual images **may** support additional sample counts, which are queried using `vkGetPhysicalDeviceImageFormatProperties` as described in [Supported Sample Counts](#).

Bits which **may** be set in the sample count limits returned by `VkPhysicalDeviceLimits`, as well as in other queries and structures representing image sample counts, are:

```

typedef enum VkSampleCountFlagBits {
    VK_SAMPLE_COUNT_1_BIT = 0x00000001,
    VK_SAMPLE_COUNT_2_BIT = 0x00000002,
    VK_SAMPLE_COUNT_4_BIT = 0x00000004,
    VK_SAMPLE_COUNT_8_BIT = 0x00000008,
    VK_SAMPLE_COUNT_16_BIT = 0x00000010,
    VK_SAMPLE_COUNT_32_BIT = 0x00000020,
    VK_SAMPLE_COUNT_64_BIT = 0x00000040,
    VK_SAMPLE_COUNT_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkSampleCountFlagBits;

```

- **VK\_SAMPLE\_COUNT\_1\_BIT** specifies an image with one sample per pixel.
- **VK\_SAMPLE\_COUNT\_2\_BIT** specifies an image with 2 samples per pixel.
- **VK\_SAMPLE\_COUNT\_4\_BIT** specifies an image with 4 samples per pixel.
- **VK\_SAMPLE\_COUNT\_8\_BIT** specifies an image with 8 samples per pixel.
- **VK\_SAMPLE\_COUNT\_16\_BIT** specifies an image with 16 samples per pixel.
- **VK\_SAMPLE\_COUNT\_32\_BIT** specifies an image with 32 samples per pixel.
- **VK\_SAMPLE\_COUNT\_64\_BIT** specifies an image with 64 samples per pixel.

```

typedef VkFlags VkSampleCountFlags;

```

**VkSampleCountFlags** is a bitmask type for setting a mask of zero or more **VkSampleCountFlagBits**.

The **VkPhysicalDevicePushDescriptorPropertiesKHR** structure is defined as:

```

typedef struct VkPhysicalDevicePushDescriptorPropertiesKHR {
    VkStructureType    sType;
    void*            pNext;
    uint32_t          maxPushDescriptors;
} VkPhysicalDevicePushDescriptorPropertiesKHR;

```

The members of the **VkPhysicalDevicePushDescriptorPropertiesKHR** structure describe the following implementation-dependent limits:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **maxPushDescriptors** is the maximum number of descriptors that **can** be used in a descriptor set created with **VK\_DESCRIPTOR\_SET\_LAYOUT\_CREATE\_PUSH\_DESCRIPTOR\_BIT\_KHR** set.

If the **VkPhysicalDevicePushDescriptorPropertiesKHR** structure is included in the **pNext** chain of **VkPhysicalDeviceProperties2**, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PUSH_DESCRIPTOR_PROPERTIES_KHR`

The `VkPhysicalDeviceMultiviewProperties` structure is defined as:

```
typedef struct VkPhysicalDeviceMultiviewProperties {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            maxMultiviewViewCount;
    uint32_t            maxMultiviewInstanceIndex;
} VkPhysicalDeviceMultiviewProperties;
```

or the equivalent

```
typedef VkPhysicalDeviceMultiviewProperties VkPhysicalDeviceMultiviewPropertiesKHR;
```

The members of the `VkPhysicalDeviceMultiviewProperties` structure describe the following implementation-dependent limits:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to an extension-specific structure.
- **maxMultiviewViewCount** is one greater than the maximum view index that **can** be used in a subpass.
- **maxMultiviewInstanceIndex** is the maximum valid value of instance index allowed to be generated by a drawing command recorded within a subpass of a multiview render pass instance.

If the `VkPhysicalDeviceMultiviewProperties` structure is included in the **pNext** chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES`

The members of the `VkPhysicalDeviceFloatControlsPropertiesKHR` structure describe the following implementation-dependent limits:

```

typedef struct VkPhysicalDeviceFloatControlsPropertiesKHR {
    VkStructureType sType;
    void* pNext;
    VkShaderFloatControlsIndependenceKHR denormBehaviorIndependence;
    VkShaderFloatControlsIndependenceKHR roundingModeIndependence;
    VkBool32 shaderSignedZeroInfNanPreserveFloat16;
    VkBool32 shaderSignedZeroInfNanPreserveFloat32;
    VkBool32 shaderSignedZeroInfNanPreserveFloat64;
    VkBool32 shaderDenormPreserveFloat16;
    VkBool32 shaderDenormPreserveFloat32;
    VkBool32 shaderDenormPreserveFloat64;
    VkBool32 shaderDenormFlushToZeroFloat16;
    VkBool32 shaderDenormFlushToZeroFloat32;
    VkBool32 shaderDenormFlushToZeroFloat64;
    VkBool32 shaderRoundingModeRTEFloat16;
    VkBool32 shaderRoundingModeRTEFloat32;
    VkBool32 shaderRoundingModeRTEFloat64;
    VkBool32 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
} VkPhysicalDeviceFloatControlsPropertiesKHR;

```

- **denormBehaviorIndependence** is a [VkShaderFloatControlsIndependenceKHR](#) value indicating whether, and how, denorm behavior can be set independently for different bit widths.
- **roundingModeIndependence** is a [VkShaderFloatControlsIndependenceKHR](#) value indicating whether, and how, rounding modes can be set independently for different bit widths.
- **shaderSignedZeroInfNanPreserveFloat16** is a boolean value indicating whether sign of a zero, Nans and  $\pm\infty$  **can** be preserved in 16-bit floating-point computations. It also indicates whether the [SignedZeroInfNanPreserve](#) execution mode **can** be used for 16-bit floating-point types.
- **shaderSignedZeroInfNanPreserveFloat32** is a boolean value indicating whether sign of a zero, Nans and  $\pm\infty$  **can** be preserved in 32-bit floating-point computations. It also indicates whether the [SignedZeroInfNanPreserve](#) execution mode **can** be used for 32-bit floating-point types.
- **shaderSignedZeroInfNanPreserveFloat64** is a boolean value indicating whether sign of a zero, Nans and  $\pm\infty$  **can** be preserved in 64-bit floating-point computations. It also indicates whether the [SignedZeroInfNanPreserve](#) execution mode **can** be used for 64-bit floating-point types.
- **shaderDenormPreserveFloat16** is a boolean value indicating whether denormals **can** be preserved in 16-bit floating-point computations. It also indicates whether the [DenormPreserve](#) execution mode **can** be used for 16-bit floating-point types.
- **shaderDenormPreserveFloat32** is a boolean value indicating whether denormals **can** be preserved in 32-bit floating-point computations. It also indicates whether the [DenormPreserve](#) execution mode **can** be used for 32-bit floating-point types.
- **shaderDenormPreserveFloat64** is a boolean value indicating whether denormals **can** be preserved in 64-bit floating-point computations. It also indicates whether the [DenormPreserve](#) execution mode **can** be used for 64-bit floating-point types.
- **shaderDenormFlushToZeroFloat16** is a boolean value indicating whether denormals **can** be flushed

to zero in 16-bit floating-point computations. It also indicates whether the `DenormFlushToZero` execution mode **can** be used for 16-bit floating-point types.

- `shaderDenormFlushToZeroFloat32` is a boolean value indicating whether denormals **can** be flushed to zero in 32-bit floating-point computations. It also indicates whether the `DenormFlushToZero` execution mode **can** be used for 32-bit floating-point types.
- `shaderDenormFlushToZeroFloat64` is a boolean value indicating whether denormals **can** be flushed to zero in 64-bit floating-point computations. It also indicates whether the `DenormFlushToZero` execution mode **can** be used for 64-bit floating-point types.
- `shaderRoundingModeRTEFloat16` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTE` execution mode **can** be used for 16-bit floating-point types.
- `shaderRoundingModeRTEFloat32` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTE` execution mode **can** be used for 32-bit floating-point types.
- `shaderRoundingModeRTEFloat64` is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTE` execution mode **can** be used for 64-bit floating-point types.
- `shaderRoundingModeRTZFloat16` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode **can** be used for 16-bit floating-point types.
- `shaderRoundingModeRTZFloat32` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode **can** be used for 32-bit floating-point types.
- `shaderRoundingModeRTZFloat64` is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the `RoundingModeRTZ` execution mode **can** be used for 64-bit floating-point types.

If the `VkPhysicalDeviceFloatControlsPropertiesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES_KHR`

Values which **may** be returned in the `denormBehaviorIndependence` and `roundingModeIndependence` fields of `VkPhysicalDeviceFloatControlsPropertiesKHR` are:

```

typedef enum VkShaderFloatControlsIndependenceKHR {
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY_KHR = 0,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL_KHR = 1,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE_KHR = 2,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_MAX_ENUM_KHR = 0x7FFFFFFF
} VkShaderFloatControlsIndependenceKHR;

```

- `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY_KHR` specifies that shader float controls for 32-bit floating point **can** be set independently; other bit widths **must** be set identically to each other.
- `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL_KHR` specifies that shader float controls for all bit widths **can** be set independently.
- `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE_KHR` specifies that shader float controls for all bit widths **must** be set identically.

The `VkPhysicalDeviceDiscardRectanglePropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceDiscardRectanglePropertiesEXT {
    VkStructureType      sType;
    void*                pNext;
    uint32_t              maxDiscardRectangles;
} VkPhysicalDeviceDiscardRectanglePropertiesEXT;

```

The members of the `VkPhysicalDeviceDiscardRectanglePropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is **NULL** or a pointer to an extension-specific structure.
- `maxDiscardRectangles` is the maximum number of active discard rectangles that **can** be specified.

If the `VkPhysicalDeviceDiscardRectanglePropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT`

The `VkPhysicalDeviceSampleLocationsPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceSampleLocationsPropertiesEXT {
    VkStructureType      sType;
    void*              pNext;
    VkSampleCountFlags   sampleLocationSampleCounts;
    VkExtent2D           maxSampleLocationGridSize;
    float              sampleLocationCoordinateRange[2];
    uint32_t             sampleLocationSubPixelBits;
    VkBool32             variableSampleLocations;
} VkPhysicalDeviceSampleLocationsPropertiesEXT;

```

The members of the `VkPhysicalDeviceSampleLocationsPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `sampleLocationSampleCounts` is a bitmask of `VkSampleCountFlagBits` indicating the sample counts supporting custom sample locations.
- `maxSampleLocationGridSize` is the maximum size of the pixel grid in which sample locations **can** vary that is supported for all sample counts in `sampleLocationSampleCounts`.
- `sampleLocationCoordinateRange[2]` is the range of supported sample location coordinates.
- `sampleLocationSubPixelBits` is the number of bits of subpixel precision for sample locations.
- `variableSampleLocations` specifies whether the sample locations used by all pipelines that will be bound to a command buffer during a subpass **must** match. If set to `VK_TRUE`, the implementation supports variable sample locations in a subpass. If set to `VK_FALSE`, then the sample locations **must** stay constant in each subpass.

If the `VkPhysicalDeviceSampleLocationsPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT`

The `VkPhysicalDeviceExternalMemoryHostPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceExternalMemoryHostPropertiesEXT {
    VkStructureType      sType;
    void*              pNext;
    VkDeviceSize         minImportedHostPointerAlignment;
} VkPhysicalDeviceExternalMemoryHostPropertiesEXT;

```

The members of the `VkPhysicalDeviceExternalMemoryHostPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `minImportedHostPointerAlignment` is the minimum **required** alignment, in bytes, for the base address and size of host pointers that **can** be imported to a Vulkan memory object.

If the `VkPhysicalDeviceExternalMemoryHostPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT`

The `VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX` structure is defined as:

```
typedef struct VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX {
    VkStructureType      sType;
    void*                pNext;
    VkBool32              perViewPositionAllComponents;
} VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX;
```

The members of the `VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `perViewPositionAllComponents` is `VK_TRUE` if the implementation supports per-view position values that differ in components other than the X component.

If the `VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PER_VIEW_ATTRIBUTES_PROPERTIES_NVX`

The `VkPhysicalDevicePointClippingProperties` structure is defined as:

```
typedef struct VkPhysicalDevicePointClippingProperties {
    VkStructureType      sType;
    void*                pNext;
    VkPointClippingBehavior pointClippingBehavior;
} VkPhysicalDevicePointClippingProperties;
```

or the equivalent

```
typedef VkPhysicalDevicePointClippingProperties  
VkPhysicalDevicePointClippingPropertiesKHR;
```

The members of the `VkPhysicalDevicePointClippingProperties` structure describe the following implementation-dependent limit:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pointClippingBehavior` is a `VkPointClippingBehavior` value specifying the point clipping behavior supported by the implementation.

If the `VkPhysicalDevicePointClippingProperties` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES`

The `VkPhysicalDeviceSubgroupProperties` structure is defined as:

```
typedef struct VkPhysicalDeviceSubgroupProperties {  
    VkStructureType          sType;  
    void*                   pNext;  
    uint32_t                subgroupSize;  
    VkShaderStageFlags       supportedStages;  
    VkSubgroupFeatureFlags  supportedOperations;  
    VkBool32                 quadOperationsInAllStages;  
} VkPhysicalDeviceSubgroupProperties;
```

The members of the `VkPhysicalDeviceSubgroupProperties` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `subgroupSize` is the default number of invocations in each subgroup. `subgroupSize` is at least 1 if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `subgroupSize` is a power-of-two.
- `supportedStages` is a bitfield of `VkShaderStageFlagBits` describing the shader stages that subgroup operations are supported in. `supportedStages` will have the `VK_SHADER_STAGE_COMPUTE_BIT` bit set if any of the physical device's queues support `VK_QUEUE_COMPUTE_BIT`.
- `supportedOperations` is a bitmask of `VkSubgroupFeatureFlagBits` specifying the sets of subgroup

operations supported on this device. `supportedOperations` will have the `VK_SUBGROUP_FEATURE_BASIC_BIT` bit set if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`.

- `quadOperationsInAllStages` is a boolean specifying whether `quad subgroup operations` are available in all stages, or are restricted to fragment and compute stages.

If the `VkPhysicalDeviceSubgroupProperties` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES`

Bits which can be set in `VkPhysicalDeviceSubgroupProperties::supportedOperations` to specify supported subgroup operations are:

```
typedef enum VkSubgroupFeatureFlagBits {
    VK_SUBGROUP_FEATURE_BASIC_BIT = 0x00000001,
    VK_SUBGROUP_FEATURE_VOTE_BIT = 0x00000002,
    VK_SUBGROUP_FEATURE_ARITHMETIC_BIT = 0x00000004,
    VK_SUBGROUP_FEATURE_BALLOT_BIT = 0x00000008,
    VK_SUBGROUP_FEATURE_SHUFFLE_BIT = 0x00000010,
    VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT = 0x00000020,
    VK_SUBGROUP_FEATURE_CLUSTERED_BIT = 0x00000040,
    VK_SUBGROUP_FEATURE_QUAD_BIT = 0x00000080,
    VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV = 0x00000100,
    VK_SUBGROUP_FEATURE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkSubgroupFeatureFlagBits;
```

- `VK_SUBGROUP_FEATURE_BASIC_BIT` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniform` capability.
- `VK_SUBGROUP_FEATURE_VOTE_BIT` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniformVote` capability.
- `VK_SUBGROUP_FEATURE_ARITHMETIC_BIT` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniformArithmetic` capability.
- `VK_SUBGROUP_FEATURE_BALLOT_BIT` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniformBallot` capability.
- `VK_SUBGROUP_FEATURE_SHUFFLE_BIT` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniformShuffle` capability.
- `VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniformShuffleRelative` capability.
- `VK_SUBGROUP_FEATURE_CLUSTERED_BIT` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniformClustered` capability.
- `VK_SUBGROUP_FEATURE_QUAD_BIT` specifies the device will accept SPIR-V shader modules containing

the `GroupNonUniformQuad` capability.

- `VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV` specifies the device will accept SPIR-V shader modules containing the `GroupNonUniformPartitionedNV` capability.

```
typedef VkFlags VkSubgroupFeatureFlags;
```

`VkSubgroupFeatureFlags` is a bitmask type for setting a mask of zero or more `VkSubgroupFeatureFlagBits`.

The `VkPhysicalDeviceSubgroupSizeControlPropertiesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceSubgroupSizeControlPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t minSubgroupSize;
    uint32_t maxSubgroupSize;
    uint32_t maxComputeWorkgroupSubgroups;
    VkShaderStageFlags requiredSubgroupSizeStages;
} VkPhysicalDeviceSubgroupSizeControlPropertiesEXT;
```

The members of the `VkPhysicalDeviceSubgroupSizeControlPropertiesEXT` structure describe the following properties:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `minSubgroupSize` is the minimum subgroup size supported by this device. `minSubgroupSize` is at least one if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `minSubgroupSize` is a power-of-two. `minSubgroupSize` is less than or equal to `maxSubgroupSize`. `minSubgroupSize` is less than or equal to `subgroupSize`.
- `maxSubgroupSize` is the maximum subgroup size supported by this device. `maxSubgroupSize` is at least one if any of the physical device's queues support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`. `maxSubgroupSize` is a power-of-two. `maxSubgroupSize` is greater than or equal to `minSubgroupSize`. `maxSubgroupSize` is greater than or equal to `subgroupSize`.
- `maxComputeWorkgroupSubgroups` is the maximum number of subgroups supported by the implementation within a workgroup.
- `requiredSubgroupSizeStages` is a bitfield of what shader stages support having a required subgroup size specified.

If the `VkPhysicalDeviceSubgroupSizeControlPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT`

The `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            advancedBlendMaxColorAttachments;
    VkBool32            advancedBlendIndependentBlend;
    VkBool32            advancedBlendNonPremultipliedSrcColor;
    VkBool32            advancedBlendNonPremultipliedDstColor;
    VkBool32            advancedBlendCorrelatedOverlap;
    VkBool32            advancedBlendAllOperations;
} VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT;
```

The members of the `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `advancedBlendMaxColorAttachments` is one greater than the highest color attachment index that **can** be used in a subpass, for a pipeline that uses an [advanced blend operation](#).
- `advancedBlendIndependentBlend` specifies whether advanced blend operations **can** vary per-attachment.
- `advancedBlendNonPremultipliedSrcColor` specifies whether the source color **can** be treated as non-premultiplied. If this is `VK_FALSE`, then `VkPipelineColorBlendAdvancedStateCreateInfoEXT ::srcPremultiplied` **must** be `VK_TRUE`.
- `advancedBlendNonPremultipliedDstColor` specifies whether the destination color **can** be treated as non-premultiplied. If this is `VK_FALSE`, then `VkPipelineColorBlendAdvancedStateCreateInfoEXT ::dstPremultiplied` **must** be `VK_TRUE`.
- `advancedBlendCorrelatedOverlap` specifies whether the overlap mode **can** be treated as correlated. If this is `VK_FALSE`, then `VkPipelineColorBlendAdvancedStateCreateInfoEXT ::blendOverlap` **must** be `VK_BLEND_OVERLAP_UNCORRELATED_EXT`.
- `advancedBlendAllOperations` specifies whether all advanced blend operation enums are supported. See the valid usage of `VkPipelineColorBlendAttachmentState`.

If the `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT`

The `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT {
    VkStructureType      sType;
    void*                pNext;
    uint32_t             maxVertexAttribDivisor;
} VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT;

```

The members of the `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxVertexAttribDivisor` is the maximum value of the number of instances that will repeat the value of vertex attribute data when instanced rendering is enabled.

If the `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_PROPERTIES_EXT`

The `VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT {
    VkStructureType      sType;
    void*                pNext;
    VkBool32              filterMinmaxSingleComponentFormats;
    VkBool32              filterMinmaxImageComponentMapping;
} VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT;

```

The members of the `VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `filterMinmaxSingleComponentFormats` is a boolean value indicating whether a minimum set of required formats support min/max filtering.
- `filterMinmaxImageComponentMapping` is a boolean value indicating whether the implementation supports non-identity component mapping of the image when doing min/max filtering.

If the `VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

If `filterMinmaxSingleComponentFormats` is `VK_TRUE`, the following formats **must** support the

`VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT_EXT` feature with `VK_IMAGE_TILING_OPTIMAL`, if they support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`.

- `VK_FORMAT_R8_UNORM`
- `VK_FORMAT_R8_SNORM`
- `VK_FORMAT_R16_UNORM`
- `VK_FORMAT_R16_SNORM`
- `VK_FORMAT_R16_SFLOAT`
- `VK_FORMAT_R32_SFLOAT`
- `VK_FORMAT_D16_UNORM`
- `VK_FORMAT_X8_D24_UNORM_PACK32`
- `VK_FORMAT_D32_SFLOAT`
- `VK_FORMAT_D16_UNORM_S8_UINT`
- `VK_FORMAT_D24_UNORM_S8_UINT`
- `VK_FORMAT_D32_SFLOAT_S8_UINT`

If the format is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports min/max filtering, and that min/max filtering of the depth aspect is supported when depth compare is disabled in the sampler.

If `filterMinmaxImageComponentMapping` is `VK_FALSE` the component mapping of the image view used with min/max filtering **must** have been created with the `r` component set to `VK_COMPONENT_SWIZZLE_IDENTITY`. Only the `r` component of the sampled image value is defined and the other component values are undefined. If `filterMinmaxImageComponentMapping` is `VK_TRUE` this restriction does not apply and image component mapping works as normal.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES_EXT`

The `VkPhysicalDeviceProtectedMemoryProperties` structure is defined as:

```
typedef struct VkPhysicalDeviceProtectedMemoryProperties {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           protectedNoFault;
} VkPhysicalDeviceProtectedMemoryProperties;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `protectedNoFault` specifies the behavior of the implementation when **protected memory access rules** are broken. If `protectedNoFault` is `VK_TRUE`, breaking those rules will not result in process termination or device loss.

If the `VkPhysicalDeviceProtectedMemoryProperties` structure is included in the `pNext` chain of

`VkPhysicalDeviceProperties2`, it is filled with a value indicating the implementation-dependent behavior.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES`

The `VkPhysicalDeviceMaintenance3Properties` structure is defined as:

```
typedef struct VkPhysicalDeviceMaintenance3Properties {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            maxPerSetDescriptors;
    VkDeviceSize        maxMemoryAllocationSize;
} VkPhysicalDeviceMaintenance3Properties;
```

or the equivalent

```
typedef VkPhysicalDeviceMaintenance3Properties
VkPhysicalDeviceMaintenance3PropertiesKHR;
```

The members of the `VkPhysicalDeviceMaintenance3Properties` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxPerSetDescriptors` is a maximum number of descriptors (summed over all descriptor types) in a single descriptor set that is guaranteed to satisfy any implementation-dependent constraints on the size of a descriptor set itself. Applications can query whether a descriptor set that goes beyond this limit is supported using `vkGetDescriptorSetLayoutSupport`.
- `maxMemoryAllocationSize` is the maximum size of a memory allocation that can be created, even if there is more space available in the heap.

If the `VkPhysicalDeviceMaintenance3Properties` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES`

The `VkPhysicalDeviceMeshShaderPropertiesNV` structure is defined as:

```

typedef struct VkPhysicalDeviceMeshShaderPropertiesNV {
    VkStructureType sType;
    void* pNext;
    uint32_t maxDrawMeshTasksCount;
    uint32_t maxTaskWorkGroupInvocations;
    uint32_t maxTaskWorkGroupSize[3];
    uint32_t maxTaskTotalMemorySize;
    uint32_t maxTaskOutputCount;
    uint32_t maxMeshWorkGroupInvocations;
    uint32_t maxMeshWorkGroupSize[3];
    uint32_t maxMeshTotalMemorySize;
    uint32_t maxMeshOutputVertices;
    uint32_t maxMeshOutputPrimitives;
    uint32_t maxMeshMultiviewViewCount;
    uint32_t meshOutputPerVertexGranularity;
    uint32_t meshOutputPerPrimitiveGranularity;
} VkPhysicalDeviceMeshShaderPropertiesNV;

```

The members of the `VkPhysicalDeviceMeshShaderPropertiesNV` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxDrawMeshTasksCount` is the maximum number of local workgroups that **can** be launched by a single draw mesh tasks command. See [Programmable Mesh Shading](#).
- `maxTaskWorkGroupInvocations` is the maximum total number of task shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the `LocalSize` execution mode in shader modules or by the object decorated by the `WorkgroupSize` decoration, **must** be less than or equal to this limit.
- `maxTaskWorkGroupSize[3]` is the maximum size of a local task workgroup. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The `x`, `y`, and `z` sizes, as specified by the `LocalSize` execution mode or by the object decorated by the `WorkgroupSize` decoration in shader modules, **must** be less than or equal to the corresponding limit.
- `maxTaskTotalMemorySize` is the maximum number of bytes that the task shader can use in total for shared and output memory combined.
- `maxTaskOutputCount` is the maximum number of output tasks a single task shader workgroup can emit.
- `maxMeshWorkGroupInvocations` is the maximum total number of mesh shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the `LocalSize` execution mode in shader modules or by the object decorated by the `WorkgroupSize` decoration, **must** be less than or equal to this limit.
- `maxMeshWorkGroupSize[3]` is the maximum size of a local mesh workgroup. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The `x`, `y`, and `z` sizes, as specified by the `LocalSize` execution mode or by the object decorated by the

`WorkgroupSize` decoration in shader modules, **must** be less than or equal to the corresponding limit.

- `maxMeshTotalMemorySize` is the maximum number of bytes that the mesh shader can use in total for shared and output memory combined.
- `maxMeshOutputVertices` is the maximum number of vertices a mesh shader output can store.
- `maxMeshOutputPrimitives` is the maximum number of primitives a mesh shader output can store.
- `maxMeshMultiviewViewCount` is the maximum number of multi-view views a mesh shader can use.
- `meshOutputPerVertexGranularity` is the granularity with which mesh vertex outputs are allocated. The value can be used to compute the memory size used by the mesh shader, which must be less than or equal to `maxMeshTotalMemorySize`.
- `meshOutputPerPrimitiveGranularity` is the granularity with which mesh outputs qualified as per-primitive are allocated. The value can be used to compute the memory size used by the mesh shader, which must be less than or equal to `maxMeshTotalMemorySize`.

If the `VkPhysicalDeviceMeshShaderPropertiesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_PROPERTIES_NV`

The `VkPhysicalDeviceDescriptorIndexingPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceDescriptorIndexingPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t maxUpdateAfterBindDescriptorsInAllPools;
    VkBool32 shaderUniformBufferArrayNonUniformIndexingNative;
    VkBool32 shaderSampledImageArrayNonUniformIndexingNative;
    VkBool32 shaderStorageBufferArrayNonUniformIndexingNative;
    VkBool32 shaderStorageImageArrayNonUniformIndexingNative;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexingNative;
    VkBool32 robustBufferAccessUpdateAfterBind;
    VkBool32 quadDivergentImplicitLod;
    uint32_t maxPerStageDescriptorUpdateAfterBindSamplers;
    uint32_t maxPerStageDescriptorUpdateAfterBindUniformBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindSampledImages;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageImages;
    uint32_t maxPerStageDescriptorUpdateAfterBindInputAttachments;
    uint32_t maxPerStageUpdateAfterBindResources;
    uint32_t maxDescriptorSetUpdateAfterBindSamplers;
    uint32_t maxDescriptorSetUpdateAfterBindUniformBuffers;
    uint32_t maxDescriptorSetUpdateAfterBindUniformBuffersDynamic;
    uint32_t maxDescriptorSetUpdateAfterBindStorageBuffers;
    uint32_t maxDescriptorSetUpdateAfterBindStorageBuffersDynamic;
    uint32_t maxDescriptorSetUpdateAfterBindSampledImages;
    uint32_t maxDescriptorSetUpdateAfterBindStorageImages;
    uint32_t maxDescriptorSetUpdateAfterBindInputAttachments;
} VkPhysicalDeviceDescriptorIndexingPropertiesEXT;

```

The members of the `VkPhysicalDeviceDescriptorIndexingPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxUpdateAfterBindDescriptorsInAllPools` is the maximum number of descriptors (summed over all descriptor types) that **can** be created across all pools that are created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT` bit set. Pool creation **may** fail when this limit is exceeded, or when the space this limit represents is unable to satisfy a pool creation due to fragmentation.
- `shaderUniformBufferArrayNonUniformIndexingNative` is a boolean value indicating whether uniform buffer descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of uniform buffers **may** execute multiple times in order to access all the descriptors.
- `shaderSampledImageArrayNonUniformIndexingNative` is a boolean value indicating whether sampler and image descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of samplers or images **may** execute multiple times in order to access all the descriptors.
- `shaderStorageBufferArrayNonUniformIndexingNative` is a boolean value indicating whether

storage buffer descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage buffers **may** execute multiple times in order to access all the descriptors.

- `shaderStorageImageArrayNonUniformIndexingNative` is a boolean value indicating whether storage image descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage images **may** execute multiple times in order to access all the descriptors.
- `shaderInputAttachmentArrayNonUniformIndexingNative` is a boolean value indicating whether input attachment descriptors natively support nonuniform indexing. If this is `VK_FALSE`, then a single dynamic instance of an instruction that nonuniformly indexes an array of input attachments **may** execute multiple times in order to access all the descriptors.
- `robustBufferAccessUpdateAfterBind` is a boolean value indicating whether `robustBufferAccess` **can** be enabled in a device simultaneously with `descriptorBindingUniformBufferUpdateAfterBind`, `descriptorBindingStorageBufferUpdateAfterBind`,  
`descriptorBindingUniformTexelBufferUpdateAfterBind`, and/or  
`descriptorBindingStorageTexelBufferUpdateAfterBind`. If this is `VK_FALSE`, then either `robustBufferAccess` **must** be disabled or all of these update-after-bind features **must** be disabled.
- `quadDivergentImplicitLod` is a boolean value indicating whether implicit level of detail calculations for image operations have well-defined results when the image and/or sampler objects used for the instruction are not uniform within a quad. See [Derivative Image Operations](#).
- `maxPerStageDescriptorUpdateAfterBindSamplers` is similar to `maxPerStageDescriptorSamplers` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxPerStageDescriptorUpdateAfterBindUniformBuffers` is similar to `maxPerStageDescriptorUniformBuffers` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxPerStageDescriptorUpdateAfterBindStorageBuffers` is similar to `maxPerStageDescriptorStorageBuffers` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxPerStageDescriptorUpdateAfterBindSampledImages` is similar to `maxPerStageDescriptorSampledImages` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxPerStageDescriptorUpdateAfterBindStorageImages` is similar to `maxPerStageDescriptorStorageImages` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxPerStageDescriptorUpdateAfterBindInputAttachments` is similar to `maxPerStageDescriptorInputAttachments` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxPerStageUpdateAfterBindResources` is similar to `maxPerStageResources` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetUpdateAfterBindSamplers` is similar to `maxDescriptorSetSamplers` but counts

descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.

- `maxDescriptorSetUpdateAfterBindUniformBuffers` is similar to `maxDescriptorSetUniformBuffers` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetUpdateAfterBindUniformBuffersDynamic` is similar to `maxDescriptorSetUniformBuffersDynamic` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetUpdateAfterBindStorageBuffers` is similar to `maxDescriptorSetStorageBuffers` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetUpdateAfterBindStorageBuffersDynamic` is similar to `maxDescriptorSetStorageBuffersDynamic` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetUpdateAfterBindSampledImages` is similar to `maxDescriptorSetSampledImages` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetUpdateAfterBindStorageImages` is similar to `maxDescriptorSetStorageImages` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetUpdateAfterBindInputAttachments` is similar to `maxDescriptorSetInputAttachments` but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.

If the `VkPhysicalDeviceDescriptorIndexingPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES_EXT`

The `VkPhysicalDeviceInlineUniformBlockPropertiesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceInlineUniformBlockPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t maxInlineUniformBlockSize;
    uint32_t maxPerStageDescriptorInlineUniformBlocks;
    uint32_t maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks;
    uint32_t maxDescriptorSetInlineUniformBlocks;
    uint32_t maxDescriptorSetUpdateAfterBindInlineUniformBlocks;
} VkPhysicalDeviceInlineUniformBlockPropertiesEXT;
```

The members of the `VkPhysicalDeviceInlineUniformBlockPropertiesEXT` structure describe the

following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxInlineUniformBlockSize` is the maximum size in bytes of an `inline uniform block` binding.
- `maxPerStageDescriptorInlineUniformBlock` is the maximum number of inline uniform block bindings that **can** be accessible to a single shader stage in a pipeline layout. Descriptor bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` count against this limit. Only descriptor bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit.
- `maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks` is similar to `maxPerStageDescriptorInlineUniformBlocks` but counts descriptor bindings from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.
- `maxDescriptorSetInlineUniformBlocks` is the maximum number of inline uniform block bindings that **can** be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptor bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` count against this limit. Only descriptor bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set count against this limit.
- `maxDescriptorSetUpdateAfterBindInlineUniformBlocks` is similar to `maxDescriptorSetInlineUniformBlocks` but counts descriptor bindings from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT` bit set.

If the `VkPhysicalDeviceInlineUniformBlockPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES_EXT`

The `VkPhysicalDeviceConservativeRasterizationPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceConservativeRasterizationPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    float primitiveOverestimationSize;
    float maxExtraPrimitiveOverestimationSize;
    float extraPrimitiveOverestimationSizeGranularity;
    VkBool32 primitiveUnderestimation;
    VkBool32 conservativePointAndLineRasterization;
    VkBool32 degenerateTrianglesRasterized;
    VkBool32 degenerateLinesRasterized;
    VkBool32 fullyCoveredFragmentShaderInputVariable;
    VkBool32 conservativeRasterizationPostDepthCoverage;
} VkPhysicalDeviceConservativeRasterizationPropertiesEXT;

```

The members of the `VkPhysicalDeviceConservativeRasterizationPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `primitiveOverestimationSize` is the size in pixels the generating primitive is increased at each of its edges during conservative rasterization overestimation mode. Even with a size of 0.0, conservative rasterization overestimation rules still apply and if any part of the pixel rectangle is covered by the generating primitive, fragments are generated for the entire pixel. However implementations **may** make the pixel coverage area even more conservative by increasing the size of the generating primitive.
- `maxExtraPrimitiveOverestimationSize` is the maximum size in pixels of extra overestimation the implementation supports in the pipeline state. A value of 0.0 means the implementation does not support any additional overestimation of the generating primitive during conservative rasterization. A value above 0.0 allows the application to further increase the size of the generating primitive during conservative rasterization overestimation.
- `extraPrimitiveOverestimationSizeGranularity` is the granularity of extra overestimation that can be specified in the pipeline state between 0.0 and `maxExtraPrimitiveOverestimationSize` inclusive. A value of 0.0 means the implementation can use the smallest representable non-zero value in the screen space pixel fixed-point grid.
- `primitiveUnderestimation` is true if the implementation supports the `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` conservative rasterization mode in addition to `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT`. Otherwise the implementation only supports `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT`.
- `conservativePointAndLineRasterization` is true if the implementation supports conservative rasterization of point and line primitives as well as triangle primitives. Otherwise the implementation only supports triangle primitives.
- `degenerateTrianglesRasterized` is false if the implementation culls primitives generated from triangles that become zero area after they are quantized to the fixed-point rasterization pixel grid. `degenerateTrianglesRasterized` is true if these primitives are not culled and the provoking vertex attributes and depth value are used for the fragments. The primitive area calculation is

done on the primitive generated from the clipped triangle if applicable. Zero area primitives are backfacing and the application **can** enable backface culling if desired.

- **degenerateLinesRasterized** is false if the implementation culs lines that become zero length after they are quantized to the fixed-point rasterization pixel grid. **degenerateLinesRasterized** is true if zero length lines are not culled and the provoking vertex attributes and depth value are used for the fragments.
- **fullyCoveredFragmentShaderInputVariable** is true if the implementation supports the SPIR-V builtin fragment shader input variable **FullyCoveredEXT** which specifies that conservative rasterization is enabled and the fragment area is fully covered by the generating primitive.
- **conservativeRasterizationPostDepthCoverage** is true if the implementation supports conservative rasterization with the **PostDepthCoverage** execution mode enabled. When supported the **SampleMask** built-in input variable will reflect the coverage after the early per-fragment depth and stencil tests are applied even when conservative rasterization is enabled. Otherwise **PostDepthCoverage** execution mode **must** not be used when conservative rasterization is enabled.

If the **VkPhysicalDeviceConservativeRasterizationPropertiesEXT** structure is included in the **pNext** chain of **VkPhysicalDeviceProperties2**, it is filled with the implementation-dependent limits and properties.

## Valid Usage (Implicit)

- **sType** must be **VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_CONSERVATIVE\_RASTERIZATION\_PROPERTIES\_EXT**

The **VkPhysicalDeviceFragmentDensityMapPropertiesEXT** structure is defined as:

```
typedef struct VkPhysicalDeviceFragmentDensityMapPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkExtent2D         minFragmentDensityTexelSize;
    VkExtent2D         maxFragmentDensityTexelSize;
    VkBool32           fragmentDensityInvocations;
} VkPhysicalDeviceFragmentDensityMapPropertiesEXT;
```

The members of the **VkPhysicalDeviceFragmentDensityMapPropertiesEXT** structure describe the following implementation-dependent limits:

- **minFragmentDensityTexelSize** is the minimum **fragment density texel size**.
- **maxFragmentDensityTexelSize** is the maximum fragment density texel size.
- **fragmentDensityInvocations** specifies whether the implementation **may** invoke additional fragment shader invocations for each covered sample.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_PROPERTIES_EXT`

If the `VkPhysicalDeviceFragmentDensityMapPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits and properties.

The `VkPhysicalDeviceShaderCorePropertiesAMD` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderCorePropertiesAMD {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            shaderEngineCount;
    uint32_t            shaderArraysPerEngineCount;
    uint32_t            computeUnitsPerShaderArray;
    uint32_t            simdPerComputeUnit;
    uint32_t            wavefrontsPerSimd;
    uint32_t            wavefrontSize;
    uint32_t            sgprsPerSimd;
    uint32_t            minSgprAllocation;
    uint32_t            maxSgprAllocation;
    uint32_t            sgprAllocationGranularity;
    uint32_t            vgprsPerSimd;
    uint32_t            minVgprAllocation;
    uint32_t            maxVgprAllocation;
    uint32_t            vgprAllocationGranularity;
} VkPhysicalDeviceShaderCorePropertiesAMD;
```

The members of the `VkPhysicalDeviceShaderCorePropertiesAMD` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shaderEngineCount` is an unsigned integer value indicating the number of shader engines found inside the shader core of the physical device.
- `shaderArraysPerEngineCount` is an unsigned integer value indicating the number of shader arrays inside a shader engine. Each shader array has its own scan converter, set of compute units, and a render back end (color and depth buffers). Shader arrays within a shader engine share shader processor input (wave launcher) and shader export (export buffer) units. Currently, a shader engine can have one or two shader arrays.
- `computeUnitsPerShaderArray` is an unsigned integer value indicating the physical number of compute units within a shader array. The active number of compute units in a shader array **may** be lower. A compute unit houses a set of SIMDs along with a sequencer module and a local data store.
- `simdPerComputeUnit` is an unsigned integer value indicating the number of SIMDs inside a

compute unit. Each SIMD processes a single instruction at a time.

- `wavefrontSize` is an unsigned integer value indicating the maximum size of a subgroup.
- `sgprsPerSimd` is an unsigned integer value indicating the number of physical Scalar General Purpose Registers (SGPRs) per SIMD.
- `minSgprAllocation` is an unsigned integer value indicating the minimum number of SGPRs allocated for a wave.
- `maxSgprAllocation` is an unsigned integer value indicating the maximum number of SGPRs allocated for a wave.
- `sgprAllocationGranularity` is an unsigned integer value indicating the granularity of SGPR allocation for a wave.
- `vgprsPerSimd` is an unsigned integer value indicating the number of physical Vector General Purpose Registers (VGPRs) per SIMD.
- `minVgprAllocation` is an unsigned integer value indicating the minimum number of VGPRs allocated for a wave.
- `maxVgprAllocation` is an unsigned integer value indicating the maximum number of VGPRs allocated for a wave.
- `vgprAllocationGranularity` is an unsigned integer value indicating the granularity of VGPR allocation for a wave.

If the `VkPhysicalDeviceShaderCorePropertiesAMD` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_AMD`

The `VkPhysicalDeviceShaderCoreProperties2AMD` structure is defined as:

```
typedef struct VkPhysicalDeviceShaderCoreProperties2AMD {
    VkStructureType           sType;
    void*                     pNext;
    VkShaderCorePropertiesFlagsAMD shaderCoreFeatures;
    uint32_t                  activeComputeUnitCount;
} VkPhysicalDeviceShaderCoreProperties2AMD;
```

The members of the `VkPhysicalDeviceShaderCoreProperties2AMD` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shaderCoreFeatures` is a bitmask of `VkShaderCorePropertiesFlagBitsAMD` indicating the set of features supported by the shader core.

- `activeComputeUnitCount` is an unsigned integer value indicating the number of compute units that have been enabled.

If the `VkPhysicalDeviceShaderCoreProperties2AMD` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_2_AMD`

Bits for this type **may** be defined by future extensions, or new versions of the `VK_AMD_shader_core_properties2` extension. Possible values of the `flags` member of `VkShaderCorePropertiesFlagsAMD` are:

```
typedef enum VkShaderCorePropertiesFlagBitsAMD {
    VK_SHADER_CORE_PROPERTIES_FLAG_BITS_MAX_ENUM_AMD = 0x7FFFFFFF
} VkShaderCorePropertiesFlagBitsAMD;
```

```
typedef VkFlags VkShaderCorePropertiesFlagsAMD;
```

`VkShaderCorePropertiesFlagsAMD` is a bitmask type for providing zero or more `VkShaderCorePropertiesFlagBitsAMD`.

The `VkPhysicalDeviceDepthStencilResolvePropertiesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceDepthStencilResolvePropertiesKHR {
    VkStructureType          sType;
    void*                   pNext;
    VkResolveModeFlagsKHR   supportedDepthResolveModes;
    VkResolveModeFlagsKHR   supportedStencilResolveModes;
    VkBool32                independentResolveNone;
    VkBool32                independentResolve;
} VkPhysicalDeviceDepthStencilResolvePropertiesKHR;
```

The members of the `VkPhysicalDeviceDepthStencilResolvePropertiesKHR` structure describe the following implementation-dependent limits:

- `supportedDepthResolveModes` is a bitmask of `VkResolveModeFlagBitsKHR` indicating the set of supported depth resolve modes. `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT_KHR` must be included in the set but implementations **may** support additional modes.
- `supportedStencilResolveModes` is a bitmask of `VkResolveModeFlagBitsKHR` indicating the set of supported stencil resolve modes. `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT_KHR` must be included in the set but implementations **may** support additional modes. `VK_RESOLVE_MODE_AVERAGE_BIT_KHR` must not be included in the set.

- `independentResolveNone` is `VK_TRUE` if the implementation supports setting the depth and stencil resolve modes to different values when one of those modes is `VK_RESOLVE_MODE_NONE_KHR`. Otherwise the implementation only supports setting both modes to the same value.
- `independentResolve` is `VK_TRUE` if the implementation supports all combinations of the supported depth and stencil resolve modes, including setting either depth or stencil resolve mode to `VK_RESOLVE_MODE_NONE_KHR`. An implementation that supports `independentResolve` **must** also support `independentResolveNone`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES_KHR`

The `VkPhysicalDevicePerformanceQueryFeaturesKHR` structure is defined as:

```
typedef struct VkPhysicalDevicePerformanceQueryFeaturesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkBool32            performanceCounterQueryPools;
    VkBool32            performanceCounterMultipleQueryPools;
} VkPhysicalDevicePerformanceQueryFeaturesKHR;
```

The members of the `VkPhysicalDevicePerformanceQueryFeaturesKHR` structure describe the following implementation-dependent features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `performanceCounterQueryPools` is `VK_TRUE` if the physical device supports performance counter query pools.
- `performanceCounterMultipleQueryPools` is `VK_TRUE` if the physical device supports using multiple performance query pools in a primary command buffer and secondary command buffers executed within it.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR`

To query performance counter query pool features supported call `vkGetPhysicalDeviceFeatures2` with a `VkPhysicalDevicePerformanceQueryFeaturesKHR` structure included in the `pNext` chain of its `pNext` parameter. The `VkPhysicalDevicePerformanceQueryFeaturesKHR` structure **can** also be in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which additional features are enabled in the device.

The `VkPhysicalDevicePerformanceQueryPropertiesKHR` structure is defined as:

```

typedef struct VkPhysicalDevicePerformanceQueryPropertiesKHR {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           allowCommandBufferQueryCopies;
} VkPhysicalDevicePerformanceQueryPropertiesKHR;

```

The members of the `VkPhysicalDevicePerformanceQueryPropertiesKHR` structure describe the following implementation-dependent properties:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `allowCommandBufferQueryCopies` is `VK_TRUE` if the performance query pools are allowed to be used with `vkCmdCopyQueryPoolResults`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR`

If the `VkPhysicalDevicePerformanceQueryPropertiesKHR` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent properties.

The `VkPhysicalDeviceShadingRateImagePropertiesNV` structure is defined as:

```

typedef struct VkPhysicalDeviceShadingRateImagePropertiesNV {
    VkStructureType    sType;
    void*             pNext;
    VkExtent2D         shadingRateTexelSize;
    uint32_t           shadingRatePaletteSize;
    uint32_t           shadingRateMaxCoarseSamples;
} VkPhysicalDeviceShadingRateImagePropertiesNV;

```

The members of the `VkPhysicalDeviceShadingRateImagePropertiesNV` structure describe the following implementation-dependent properties related to the `shading rate image` feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shadingRateTexelSize` indicates the width and height of the portion of the framebuffer corresponding to each texel in the shading rate image.
- `shadingRatePaletteSize` indicates the maximum number of palette entries supported for the shading rate image.
- `shadingRateMaxCoarseSamples` specifies the maximum number of coverage samples supported in a single fragment. If the product of the fragment size derived from the base shading rate and the number of coverage samples per pixel exceeds this limit, the final shading rate will be adjusted so that its product does not exceed the limit.

If the `VkPhysicalDeviceShadingRateImagePropertiesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_PROPERTIES_NV`

The `VkPhysicalDeviceTransformFeedbackPropertiesEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceTransformFeedbackPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            maxTransformFeedbackStreams;
    uint32_t            maxTransformFeedbackBuffers;
    VkDeviceSize        maxTransformFeedbackBufferSize;
    uint32_t            maxTransformFeedbackStreamDataSize;
    uint32_t            maxTransformFeedbackBufferDataSize;
    uint32_t            maxTransformFeedbackBufferDataStride;
    VkBool32            transformFeedbackQueries;
    VkBool32            transformFeedbackStreamsLinesTriangles;
    VkBool32            transformFeedbackRasterizationStreamSelect;
    VkBool32            transformFeedbackDraw;
} VkPhysicalDeviceTransformFeedbackPropertiesEXT;
```

The members of the `VkPhysicalDeviceTransformFeedbackPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxTransformFeedbackStreams` is the maximum number of vertex streams that can be output from geometry shaders declared with the `GeometryStreams` capability. If the implementation does not support `VkPhysicalDeviceTransformFeedbackFeaturesEXT::geometryStreams` then `maxTransformFeedbackStreams` must be set to 1.
- `maxTransformFeedbackBuffers` is the maximum number of transform feedback buffers that can be bound for capturing shader outputs from the last vertex processing stage.
- `maxTransformFeedbackBufferSize` is the maximum size that can be specified when binding a buffer for transform feedback in `vkCmdBindTransformFeedbackBuffersEXT`.
- `maxTransformFeedbackStreamDataSize` is the maximum amount of data in bytes for each vertex that captured to one or more transform feedback buffers associated with a specific vertex stream.
- `maxTransformFeedbackBufferDataSize` is the maximum amount of data in bytes for each vertex that can be captured to a specific transform feedback buffer.
- `maxTransformFeedbackBufferDataStride` is the maximum stride between each capture of vertex data to the buffer.

- `transformFeedbackQueries` is true if the implementation supports the `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` query type. `transformFeedbackQueries` is false if queries of this type **cannot** be created.
- `transformFeedbackStreamsLinesTriangles` is true if the implementation supports the geometry shader `OpExecutionMode` of `OutputLineStrip` and `OutputTriangleStrip` in addition to `OutputPoints` when more than one vertex stream is output. If `transformFeedbackStreamsLinesTriangles` is false the implementation only supports an `OpExecutionMode` of `OutputPoints` when more than one vertex stream is output from the geometry shader.
- `transformFeedbackRasterizationStreamSelect` is true if the implementation supports the `GeometryStreams` SPIR-V capability and the application can use `VkPipelineRasterizationStateStreamCreateInfoEXT` to modify which vertex stream output is used for rasterization. Otherwise vertex stream `0` **must** always be used for rasterization.
- `transformFeedbackDraw` is true if the implementation supports the `vkCmdDrawIndirectByteCountEXT` function otherwise the function **must** not be called.

If the `VkPhysicalDeviceTransformFeedbackPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits and properties.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_PROPERTIES_EXT`

The `VkPhysicalDeviceRayTracingPropertiesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceRayTracingPropertiesNV {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            shaderGroupHandleSize;
    uint32_t            maxRecursionDepth;
    uint32_t            maxShaderGroupStride;
    uint32_t            shaderGroupBaseAlignment;
    uint64_t            maxGeometryCount;
    uint64_t            maxInstanceCount;
    uint64_t            maxTriangleCount;
    uint32_t            maxDescriptorSetAccelerationStructures;
} VkPhysicalDeviceRayTracingPropertiesNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shaderGroupHandleSize` size in bytes of the shader header.
- `maxRecursionDepth` is the maximum number of levels of recursion allowed in a trace command.
- `maxShaderGroupStride` is the maximum stride in bytes allowed between shader groups in the SBT.
- `shaderGroupBaseAlignment` is the required alignment in bytes for the base of the SBTs.

- `maxGeometryCount` is the maximum number of geometries in the bottom level acceleration structure.
- `maxInstanceCount` is the maximum number of instances in the top level acceleration structure.
- `maxTriangleCount` is the maximum number of triangles in all geometries in the bottom level acceleration structure.
- `maxDescriptorSetAccelerationStructures` is the maximum number of acceleration structure descriptors that are allowed in a descriptor set.

If the `VkPhysicalDeviceRayTracingPropertiesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PROPERTIES_NV`

The `VkPhysicalDeviceCooperativeMatrixPropertiesNV` structure is defined as:

```
typedef struct VkPhysicalDeviceCooperativeMatrixPropertiesNV {
    VkStructureType      sType;
    void*                pNext;
    VkShaderStageFlags   cooperativeMatrixSupportedStages;
} VkPhysicalDeviceCooperativeMatrixPropertiesNV;
```

The members of the `VkPhysicalDeviceCooperativeMatrixPropertiesNV` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `cooperativeMatrixSupportedStages` is a bitfield of `VkShaderStageFlagBits` describing the shader stages that cooperative matrix instructions are supported in. `cooperativeMatrixSupportedStages` will have the `VK_SHADER_STAGE_COMPUTE_BIT` bit set if any of the physical device's queues support `VK_QUEUE_COMPUTE_BIT`.

If the `VkPhysicalDeviceCooperativeMatrixPropertiesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_PROPERTIES_NV`

The `VkPhysicalDeviceShaderSMBuiltinsPropertiesNV` structure is defined as:

```

typedef struct VkPhysicalDeviceShaderSMBuiltinsPropertiesNV {
    VkStructureType    sType;
    void*             pNext;
    uint32_t           shaderSMCount;
    uint32_t           shaderWarpsPerSM;
} VkPhysicalDeviceShaderSMBuiltinsPropertiesNV;

```

The members of the `VkPhysicalDeviceShaderSMBuiltinsPropertiesNV` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `shaderSMCount` is the number of SMs on the device.
- `shaderWarpsPerSM` is the maximum number of simultaneously executing warps on an SM.

If the `VkPhysicalDeviceShaderSMBuiltinsPropertiesNV` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_PROPERTIES_NV`

The `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkDeviceSize       storageTexelBufferOffsetAlignmentBytes;
    VkBool32           storageTexelBufferOffsetSingleTexelAlignment;
    VkDeviceSize       uniformTexelBufferOffsetAlignmentBytes;
    VkBool32           uniformTexelBufferOffsetSingleTexelAlignment;
} VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT;

```

The members of the `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `storageTexelBufferOffsetAlignmentBytes` is a byte alignment that is sufficient for a storage texel buffer of any format.
- `storageTexelBufferOffsetSingleTexelAlignment` indicates whether single texel alignment is sufficient for a storage texel buffer of any format.
- `uniformTexelBufferOffsetAlignmentBytes` is a byte alignment that is sufficient for a uniform texel buffer of any format.

- `uniformTexelBufferOffsetSingleTexelAlignment` indicates whether single texel alignment is sufficient for a uniform texel buffer of any format.

If the `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

If the single texel alignment property is `VK_FALSE`, then the buffer view's offset **must** be aligned to the corresponding byte alignment value. If the single texel alignment property is `VK_TRUE`, then the buffer view's offset **must** be aligned to the lesser of the corresponding byte alignment value or the size of a single texel, based on `VkBufferViewCreateInfo::format`. If the size of a single texel is a multiple of three bytes, then the size of a single component of the format is used instead.

These limits **must** not advertise a larger alignment than the `required` maximum minimum value of `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`, for any format that supports use as a texel buffer.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT`

To query the timeline semaphore properties of a physical device, add `VkPhysicalDeviceTimelineSemaphorePropertiesKHR` to the `pNext` chain of the `VkPhysicalDeviceProperties2` structure. The `VkPhysicalDeviceTimelineSemaphorePropertiesKHR` structure is defined as:

```
typedef struct VkPhysicalDeviceTimelineSemaphorePropertiesKHR {
    VkStructureType sType;
    void* pNext;
    uint64_t maxTimelineSemaphoreValueDifference;
} VkPhysicalDeviceTimelineSemaphorePropertiesKHR;
```

The members of the `VkPhysicalDeviceTimelineSemaphorePropertiesKHR` structure describe the following implementation-dependent limits:

- `maxTimelineSemaphoreValueDifference` indicates the maximum difference allowed by the implementation between the current value of a timeline semaphore and any pending signal or wait operations.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES_KHR`

The `VkPhysicalDeviceLineRasterizationPropertiesEXT` structure is defined as:

```

typedef struct VkPhysicalDeviceLineRasterizationPropertiesEXT {
    VkStructureType    sType;
    void*             pNext;
    uint32_t          lineSubPixelPrecisionBits;
} VkPhysicalDeviceLineRasterizationPropertiesEXT;

```

The members of the `VkPhysicalDeviceLineRasterizationPropertiesEXT` structure describe the following implementation-dependent limits:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `lineSubPixelPrecisionBits` is the number of bits of subpixel precision in framebuffer coordinates  $x_f$  and  $y_f$  when rasterizing [line segments](#).

If the `VkPhysicalDeviceLineRasterizationPropertiesEXT` structure is included in the `pNext` chain of `VkPhysicalDeviceProperties2`, it is filled with the implementation-dependent limits.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_PROPERTIES_EXT`

## 36.1. Limit Requirements

The following table specifies the **required** minimum/maximum for all Vulkan graphics implementations. Where a limit corresponds to a fine-grained device feature which is **optional**, the feature name is listed with two **required** limits, one when the feature is supported and one when it is not supported. If an implementation supports a feature, the limits reported are the same whether or not the feature is enabled.

*Table 45. Required Limit Types*

Type	Limit	Feature
<code>uint32_t</code>	<code>maxImageDimension1D</code>	-
<code>uint32_t</code>	<code>maxImageDimension2D</code>	-
<code>uint32_t</code>	<code>maxImageDimension3D</code>	-
<code>uint32_t</code>	<code>maxImageDimensionCube</code>	-
<code>uint32_t</code>	<code>maxImageArrayLayers</code>	-
<code>uint32_t</code>	<code>maxTexelBufferElements</code>	-
<code>uint32_t</code>	<code>maxUniformBufferRange</code>	-
<code>uint32_t</code>	<code>maxStorageBufferRange</code>	-
<code>uint32_t</code>	<code>maxPushConstantsSize</code>	-
<code>uint32_t</code>	<code>maxMemoryAllocationCount</code>	-
<code>uint32_t</code>	<code>maxSamplerAllocationCount</code>	-

Type	Limit	Feature
VkDeviceSize	bufferImageGranularity	-
VkDeviceSize	sparseAddressSpaceSize	sparseBinding
uint32_t	maxBoundDescriptorSets	-
uint32_t	maxPerStageDescriptorSamplers	-
uint32_t	maxPerStageDescriptorUniformBuffers	-
uint32_t	maxPerStageDescriptorStorageBuffers	-
uint32_t	maxPerStageDescriptorSampledImages	-
uint32_t	maxPerStageDescriptorStorageImages	-
uint32_t	maxPerStageDescriptorInputAttachments	-
uint32_t	maxPerStageResources	-
uint32_t	maxDescriptorSetSamplers	-
uint32_t	maxDescriptorSetUniformBuffers	-
uint32_t	maxDescriptorSetUniformBuffersDynamic	-
uint32_t	maxDescriptorSetStorageBuffers	-
uint32_t	maxDescriptorSetStorageBuffersDynamic	-
uint32_t	maxDescriptorSetSampledImages	-
uint32_t	maxDescriptorSetStorageImages	-
uint32_t	maxDescriptorSetInputAttachments	-
uint32_t	maxVertexInputAttributes	-
uint32_t	maxVertexInputBindings	-
uint32_t	maxVertexInputAttributeOffset	-
uint32_t	maxVertexInputBindingStride	-
uint32_t	maxVertexOutputComponents	-
uint32_t	maxTessellationGenerationLevel	tessellationShader
uint32_t	maxTessellationPatchSize	tessellationShader
uint32_t	maxTessellationControlPerVertexInputComponents	tessellationShader
uint32_t	maxTessellationControlPerVertexOutputComponents	tessellationShader
uint32_t	maxTessellationControlPerPatchOutputComponents	tessellationShader
uint32_t	maxTessellationControlTotalOutputComponents	tessellationShader
uint32_t	maxTessellationEvaluationInputComponents	tessellationShader
uint32_t	maxTessellationEvaluationOutputComponents	tessellationShader
uint32_t	maxGeometryShaderInvocations	geometryShader
uint32_t	maxGeometryInputComponents	geometryShader
uint32_t	maxGeometryOutputComponents	geometryShader
uint32_t	maxGeometryOutputVertices	geometryShader
uint32_t	maxGeometryTotalOutputComponents	geometryShader
uint32_t	maxFragmentInputComponents	-

Type	Limit	Feature
uint32_t	maxFragmentOutputAttachments	-
uint32_t	maxFragmentDualSrcAttachments	dualSrcBlend
uint32_t	maxFragmentCombinedOutputResources	-
uint32_t	maxComputeSharedMemorySize	-
3 × uint32_t	maxComputeWorkGroupCount	-
uint32_t	maxComputeWorkGroupInvocations	-
3 × uint32_t	maxComputeWorkGroupSize	-
uint32_t	subPixelPrecisionBits	-
uint32_t	subTexelPrecisionBits	-
uint32_t	mipmapPrecisionBits	-
uint32_t	maxDrawIndexedIndexValue	fullDrawIndexUint32
uint32_t	maxDrawIndirectCount	multiDrawIndirect
float	maxSamplerLodBias	-
float	maxSamplerAnisotropy	samplerAnisotropy
uint32_t	maxViewports	multiViewport
2 × uint32_t	maxViewportDimensions	-
2 × float	viewportBoundsRange	-
uint32_t	viewportSubPixelBits	-
size_t	minMemoryMapAlignment	-
VkDeviceSize	minTexelBufferOffsetAlignment	-
VkDeviceSize	minUniformBufferOffsetAlignment	-
VkDeviceSize	minStorageBufferOffsetAlignment	-
int32_t	minTexelOffset	-
uint32_t	maxTexelOffset	-
int32_t	minTexelGatherOffset	shaderImageGatherExtended
uint32_t	maxTexelGatherOffset	shaderImageGatherExtended
float	minInterpolationOffset	sampleRateShading
float	maxInterpolationOffset	sampleRateShading
uint32_t	subPixelInterpolationOffsetBits	sampleRateShading
uint32_t	maxFramebufferWidth	-
uint32_t	maxFramebufferHeight	-
uint32_t	maxFramebufferLayers	-
VkSampleCountFlags	framebufferColorSampleCounts	-
VkSampleCountFlags	framebufferDepthSampleCounts	-
VkSampleCountFlags	framebufferStencilSampleCounts	-

Type	Limit	Feature
VkSampleCountFlags	framebufferNoAttachmentsSampleCounts	-
uint32_t	maxColorAttachments	-
VkSampleCountFlags	sampledImageColorSampleCounts	-
VkSampleCountFlags	sampledImageIntegerSampleCounts	-
VkSampleCountFlags	sampledImageDepthSampleCounts	-
VkSampleCountFlags	sampledImageStencilSampleCounts	-
VkSampleCountFlags	storageImageSampleCounts	shaderStorageImageMultisample
uint32_t	maxSampleMaskWords	-
VkBool32	timestampComputeAndGraphics	-
float	timestampPeriod	-
uint32_t	maxClipDistances	shaderClipDistance
uint32_t	maxCullDistances	shaderCullDistance
uint32_t	maxCombinedClipAndCullDistances	shaderCullDistance
uint32_t	discreteQueuePriorities	-
2 × float	pointSizeRange	largePoints
2 × float	lineWidthRange	wideLines
float	pointSizeGranularity	largePoints
float	lineWidthGranularity	wideLines
VkBool32	strictLines	-
VkBool32	standardSampleLocations	-
VkDeviceSize	optimalBufferCopyOffsetAlignment	-
VkDeviceSize	optimalBufferCopyRowPitchAlignment	-
VkDeviceSize	nonCoherentAtomSize	-
uint32_t	maxDiscardRectangles	VK_EXT_discard_rectangles
VkBool32	filterMinmaxSingleComponentFormats	VK_EXT_sampler_filter_minmax
VkBool32	filterMinmaxImageComponentMapping	VK_EXT_sampler_filter_minmax
float	primitiveOverestimationSize	VK_EXT_conservative_rasterization
VkBool32	maxExtraPrimitiveOverestimationSize	VK_EXT_conservative_rasterization
float	extraPrimitiveOverestimationSizeGranularity	VK_EXT_conservative_rasterization
VkBool32	degenerateTriangleRasterized	VK_EXT_conservative_rasterization

Type	Limit	Feature
float	degenerateLinesRasterized	VK_EXT_conservative_rasterization
VkBool32	fullyCoveredFragmentShaderInputVariable	VK_EXT_conservative_rasterization
VkBool32	conservativeRasterizationPostDepthCoverage	VK_EXT_conservative_rasterization
uint32_t	maxVertexAttribDivisor	VK_EXT_vertex_attribute_divisor
uint32_t	maxDrawMeshTasksCount	VK_NV_mesh_shader
uint32_t	maxTaskWorkGroupInvocations	VK_NV_mesh_shader
uint32_t	maxTaskWorkGroupSize	VK_NV_mesh_shader
uint32_t	maxTaskTotalMemorySize	VK_NV_mesh_shader
uint32_t	maxTaskOutputCount	VK_NV_mesh_shader
uint32_t	maxMeshWorkGroupInvocations	VK_NV_mesh_shader
uint32_t	maxMeshWorkGroupSize	VK_NV_mesh_shader
uint32_t	maxMeshTotalMemorySize	VK_NV_mesh_shader
uint32_t	maxMeshOutputVertices	VK_NV_mesh_shader
uint32_t	maxMeshOutputPrimitives	VK_NV_mesh_shader
uint32_t	maxMeshMultiviewViewCount	VK_NV_mesh_shader
uint32_t	meshOutputPerVertexGranularity	VK_NV_mesh_shader
uint32_t	meshOutputPerPrimitiveGranularity	VK_NV_mesh_shader
uint32_t	maxTransformFeedbackStreams	VK_EXT_transform_feedback
uint32_t	maxTransformFeedbackBuffers	VK_EXT_transform_feedback
VkDeviceSize	maxTransformFeedbackBufferSize	VK_EXT_transform_feedback
uint32_t	maxTransformFeedbackStreamDataSize	VK_EXT_transform_feedback
uint32_t	maxTransformFeedbackBufferDataSize	VK_EXT_transform_feedback
uint32_t	maxTransformFeedbackBufferDataStride	VK_EXT_transform_feedback
VkBool32	transformFeedbackQueries	VK_EXT_transform_feedback
VkBool32	transformFeedbackStreamsLinesTriangles	VK_EXT_transform_feedback
VkBool32	transformFeedbackRasterizationStreamSelect	VK_EXT_transform_feedback
VkBool32	transformFeedbackDraw	VK_EXT_transform_feedback
VkExtent2D	minFragmentDensityTexelSize	fragmentDensityMap
VkExtent2D	maxFragmentDensityTexelSize	fragmentDensityMap
VkBool32	fragmentDensityInvocations	fragmentDensityMap
uint32_t	shaderGroupHandleSize	VK_NV_ray_tracing
uint32_t	maxRecursionDepth	VK_NV_ray_tracing
uint32_t	shaderGroupBaseAlignment	VK_NV_ray_tracing
uint32_t	maxGeometryCount	VK_NV_ray_tracing
uint32_t	maxInstanceCount	VK_NV_ray_tracing
uint32_t	maxTriangleCount	VK_NV_ray_tracing
uint32_t	maxDescriptorSetAccelerationStructures	VK_NV_ray_tracing
uint64_t	maxTimelineSemaphoreValueDifference	timelineSemaphore
uint32_t	lineSubPixelPrecisionBits	VK_EXT_line_rasterization

Table 46. Required Limits

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
maxImageDimension1D	-	4096	min
maxImageDimension2D	-	4096	min
maxImageDimension3D	-	256	min
maxImageDimensionCube	-	4096	min
maxImageArrayLayers	-	256	min
maxTexelBufferElements	-	65536	min
maxUniformBufferRange	-	16384	min
maxStorageBufferRange	-	$2^{27}$	min
maxPushConstantsSize	-	128	min
maxMemoryAllocationCount	-	4096	min
maxSamplerAllocationCount	-	4000	min
bufferImageGranularity	-	131072	max
sparseAddressSpaceSize	0	$2^{31}$	min
maxBoundDescriptorSets	-	4	min
maxPerStageDescriptorSamplers	-	16	min
maxPerStageDescriptorUniformBuffers	-	12	min
maxPerStageDescriptorStorageBuffers	-	4	min
maxPerStageDescriptorSampledImages	-	16	min
maxPerStageDescriptorStorageImages	-	4	min
maxPerStageDescriptorInputAttachments	-	4	min
maxPerStageResources	-	128 <sup>2</sup>	min
maxDescriptorSetSamplers	-	96 <sup>8</sup>	min, n × PerStage
maxDescriptorSetUniformBuffers	-	72 <sup>8</sup>	min, n × PerStage
maxDescriptorSetUniformBuffersDynamic	-	8	min
maxDescriptorSetStorageBuffers	-	24 <sup>8</sup>	min, n × PerStage
maxDescriptorSetStorageBuffersDynamic	-	4	min
maxDescriptorSetSampledImages	-	96 <sup>8</sup>	min, n × PerStage
maxDescriptorSetStorageImages	-	24 <sup>8</sup>	min, n × PerStage
maxDescriptorSetInputAttachments	-	4	min
maxVertexInputAttributes	-	16	min
maxVertexInputBindings	-	16	min

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
maxVertexInputAttributeOffset	-	2047	min
maxVertexInputBindingStride	-	2048	min
maxVertexOutputComponents	-	64	min
maxTessellationGenerationLevel	0	64	min
maxTessellationPatchSize	0	32	min
maxTessellationControlPerVertexInputComponents	0	64	min
maxTessellationControlPerVertexOutputComponents	0	64	min
maxTessellationControlPerPatchOutputComponents	0	120	min
maxTessellationControlTotalOutputComponents	0	2048	min
maxTessellationEvaluationInputComponents	0	64	min
maxTessellationEvaluationOutputComponents	0	64	min
maxGeometryShaderInvocations	0	32	min
maxGeometryInputComponents	0	64	min
maxGeometryOutputComponents	0	64	min
maxGeometryOutputVertices	0	256	min
maxGeometryTotalOutputComponents	0	1024	min
maxFragmentInputComponents	-	64	min
maxFragmentOutputAttachments	-	4	min
maxFragmentDualSrcAttachments	0	1	min
maxFragmentCombinedOutputResources	-	4	min
maxComputeSharedMemorySize	-	16384	min
maxComputeWorkGroupCount	-	(65535,65535,65535)	min
maxComputeWorkGroupInvocations	-	128	min
maxComputeWorkGroupSize	-	(128,128,64)	min
subPixelPrecisionBits	-	4	min
subTexelPrecisionBits	-	4	min
mipmapPrecisionBits	-	4	min
maxDrawIndexedIndexValue	$2^{24}-1$	$2^{32}-1$	min
maxDrawIndirectCount	1	$2^{16}-1$	min
maxSamplerLodBias	-	2	min
maxSamplerAnisotropy	1	16	min
maxViewports	1	16	min
maxViewportDimensions	-	(4096,4096) <sup>3</sup>	min
viewportBoundsRange	-	(-8192,8191) <sup>4</sup>	(max,min)

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
viewportSubPixelBits	-	0	min
minMemoryMapAlignment	-	64	min
minTexelBufferOffsetAlignment	-	256	max
minUniformBufferOffsetAlignment	-	256	max
minStorageBufferOffsetAlignment	-	256	max
minTexelOffset	-	-8	max
maxTexelOffset	-	7	min
minTexelGatherOffset	0	-8	max
maxTexelGatherOffset	0	7	min
minInterpolationOffset	0.0	-0.5 <sup>5</sup>	max
maxInterpolationOffset	0.0	0.5 - (1 ULP) <sup>5</sup>	min
subPixelInterpolationOffsetBits	0	4 <sup>5</sup>	min
maxFramebufferWidth	-	4096	min
maxFramebufferHeight	-	4096	min
maxFramebufferLayers	-	256	min
framebufferColorSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min
framebufferDepthSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min
framebufferStencilSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min
framebufferNoAttachmentsSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min
maxColorAttachments	-	4	min
sampledImageColorSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min
sampledImageIntegerSampleCounts	-	VK_SAMPLE_COUNT_1_BIT	min
sampledImageDepthSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
sampledImageStencilSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min
storageImageSampleCounts	VK_SAMPLE_COUNT_1_BIT	(VK_SAMPLE_COUNT_1_BIT   VK_SAMPLE_COUNT_4_BIT)	min
maxSampleMaskWords	-	1	min
timestampComputeAndGraphics	-	-	implementation dependent
timestampPeriod	-	-	duration
maxClipDistances	0	8	min
maxCullDistances	0	8	min
maxCombinedClipAndCullDistances	0	8	min
discreteQueuePriorities	-	2	min
pointSizeRange	(1.0,1.0)	(1.0,64.0 - ULP) <sup>6</sup>	(max,min)
lineWidthRange	(1.0,1.0)	(1.0,8.0 - ULP) <sup>7</sup>	(max,min)
pointSizeGranularity	0.0	1.0 <sup>6</sup>	max, fixed point increment
lineWidthGranularity	0.0	1.0 <sup>7</sup>	max, fixed point increment
strictLines	-	-	implementation dependent
standardSampleLocations	-	-	implementation dependent
optimalBufferCopyOffsetAlignment	-	-	recommendation
optimalBufferCopyRowPitchAlignment	-	-	recommendation
nonCoherentAtomSize	-	256	max
maxPushDescriptors	-	32	min
maxMultiviewViewCount	-	6	min
maxMultiviewInstanceIndex	-	2 <sup>27</sup> -1	min
maxDiscardRectangles	0	4	min
sampleLocationSampleCounts	-	VK_SAMPLE_COUNT_4_BIT	min
maxSampleLocationGridSize	-	(1,1)	min

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
sampleLocationCoordinateRange	-	(0.0, 0.9375)	(max,min)
sampleLocationSubPixelBits	-	4	min
variableSampleLocations	-	false	implementation dependent
minImportedHostPointerAlignment	-	65536	max
perViewPositionAllComponents	-	-	implementation dependent
filterMinmaxSingleComponentFormats	-	-	implementation dependent
filterMinmaxImageComponentMapping	-	-	implementation dependent
advancedBlendMaxColorAttachments	-	1	min
advancedBlendIndependentBlend	-	false	implementation dependent
advancedBlendNonPremultipliedSrcColor	-	false	implementation dependent
advancedBlendNonPremultipliedDstColor	-	false	implementation dependent
advancedBlendCorrelatedOverlap	-	false	implementation dependent
advancedBlendAllOperations	-	false	implementation dependent
maxPerSetDescriptors	-	1024	min
maxMemoryAllocationSize	-	$2^{30}$	min
primitiveOverestimationSize	-	0.0	min
maxExtraPrimitiveOverestimationSize	-	0.0	min
extraPrimitiveOverestimationSizeGranularity	-	0.0	min
primitiveUnderestimation	-	false	implementation dependent
conservativePointAndLineRasterization	-	false	implementation dependent
degenerateTrianglesRasterized	-	false	implementation dependent
degenerateLinesRasterized	-	false	implementation dependent
fullyCoveredFragmentShaderInputVariable	-	false	implementation dependent
conservativeRasterizationPostDepthCoverage	-	false	implementation dependent
maxUpdateAfterBindDescriptorsInAllPools	-	500000	min

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
shaderUniformBufferArrayNonUniformIndexingNative	-	false	implementation dependent
shaderSampledImageArrayNonUniformIndexingNative	-	false	implementation dependent
shaderStorageBufferArrayNonUniformIndexingNative	-	false	implementation dependent
shaderStorageImageArrayNonUniformIndexingNative	-	false	implementation dependent
shaderInputAttachmentArrayNonUniformIndexingNative	-	false	implementation dependent
maxPerStageDescriptorUpdateAfterBindSamplers	-	500000 <sup>9</sup>	min
maxPerStageDescriptorUpdateAfterBindUniformBuffers	-	12 <sup>9</sup>	min
maxPerStageDescriptorUpdateAfterBindStorageBuffers	-	500000 <sup>9</sup>	min
maxPerStageDescriptorUpdateAfterBindSampledImages	-	500000 <sup>9</sup>	min
maxPerStageDescriptorUpdateAfterBindStorageImages	-	500000 <sup>9</sup>	min
maxPerStageDescriptorUpdateAfterBindInputAttachments	-	4 <sup>9</sup>	min
maxPerStageUpdateAfterBindResources	-	500000 <sup>9</sup>	min
maxDescriptorSetUpdateAfterBindSamplers	-	500000 <sup>9</sup>	min
maxDescriptorSetUpdateAfterBindUniformBuffers	-	72 <sup>8,9</sup>	min, n × PerStage
maxDescriptorSetUpdateAfterBindUniformBuffersDynamic	-	8 <sup>9</sup>	min
maxDescriptorSetUpdateAfterBindStorageBuffers	-	500000 <sup>9</sup>	min
maxDescriptorSetUpdateAfterBindStorageBuffersDynamic	-	4 <sup>9</sup>	min
maxDescriptorSetUpdateAfterBindSampledImages	-	500000 <sup>9</sup>	min
maxDescriptorSetUpdateAfterBindStorageImages	-	500000 <sup>9</sup>	min
maxDescriptorSetUpdateAfterBindInputAttachments	-	4 <sup>9</sup>	min
maxInlineUniformBlockSize	-	256	min
maxPerStageDescriptorInlineUniformBlocks	-	4	min
maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks	-	4	min
maxDescriptorSetInlineUniformBlocks	-	4	min
maxDescriptorSetUpdateAfterBindInlineUniformBlocks	-	4	min
maxVertexAttribDivisor	-	2 <sup>16</sup> -1	min
maxDrawMeshTasksCount	-	2 <sup>16</sup> -1	min

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
maxTaskWorkGroupInvocations	-	32	min
maxTaskWorkGroupSize	-	(32,1,1)	min
maxTaskTotalMemorySize	-	16384	min
maxTaskOutputCount	-	$2^{16}-1$	min
maxMeshWorkGroupInvocations	-	32	min
maxMeshWorkGroupSize	-	(32,1,1)	min
maxMeshTotalMemorySize	-	16384	min
maxMeshOutputVertices	-	256	min
maxMeshOutputPrimitives	-	256	min
maxMeshMultiviewViewCount	-	1	min
meshOutputPerVertexGranularity	-	-	implementation dependent
meshOutputPerPrimitiveGranularity	-	-	implementation dependent
maxTransformFeedbackStreams	-	1	min
maxTransformFeedbackBuffers	-	1	min
maxTransformFeedbackBufferSize	-	$2^{27}$	min
maxTransformFeedbackStreamDataSize	-	512	min
maxTransformFeedbackBufferDataSize	-	512	min
maxTransformFeedbackBufferDataStride	-	512	min
transformFeedbackQueries	-	false	implementation dependent
transformFeedbackStreamsLinesTriangles	-	false	implementation dependent
transformFeedbackRasterizationStreamSelect	-	false	implementation dependent
transformFeedbackDraw	-	false	implementation dependent
minFragmentDensityTexelSize	-	(1,1)	min
maxFragmentDensityTexelSize	-	(1,1)	min
fragmentDensityInvocations	-	-	implementation dependent
shaderGroupHandleSize	-	16	min
maxRecursionDepth	-	31	min
shaderGroupBaseAlignment	-	64	max
maxGeometryCount	-	$2^{24}-1$	min
maxInstanceCount	-	$2^{24}-1$	min

Limit	Unsupported Limit	Supported Limit	Limit Type <sup>1</sup>
maxTriangleCount	-	$2^{29}-1$	min
maxDescriptorSetAccelerationStructures	-	16	min
maxTimelineSemaphoreValueDifference	-	$2^{31}-1$	min
lineSubPixelPrecisionBits	-	4	min

1

The **Limit Type** column specifies the limit is either the minimum limit all implementations **must** support or the maximum limit all implementations **must** support. For bitmasks a minimum limit is the least bits all implementations **must** set, but they **may** have additional bits set beyond this minimum.

2

The `maxPerStageResources` **must** be at least the smallest of the following:

- the sum of the `maxPerStageDescriptorUniformBuffers`, `maxPerStageDescriptorStorageBuffers`, `maxPerStageDescriptorSampledImages`, `maxPerStageDescriptorStorageImages`, `maxPerStageDescriptorInputAttachments`, `maxColorAttachments` limits, or
- 128.

It **may** not be possible to reach this limit in every stage.

3

See `maxViewportDimensions` for the **required** relationship to other limits.

4

See `viewportBoundsRange` for the **required** relationship to other limits.

5

The values `minInterpolationOffset` and `maxInterpolationOffset` describe the closed interval of supported interpolation offsets:  $[\text{minInterpolationOffset}, \text{maxInterpolationOffset}]$ . The ULP is determined by `subPixelInterpolationOffsetBits`. If `subPixelInterpolationOffsetBits` is 4, this provides increments of  $(1/2^4) = 0.0625$ , and thus the range of supported interpolation offsets would be  $[-0.5, 0.4375]$ .

6

The point size ULP is determined by `pointSizeGranularity`. If the `pointSizeGranularity` is 0.125, the range of supported point sizes **must** be at least [1.0, 63.875].

7

The line width ULP is determined by `lineWidthGranularity`. If the `lineWidthGranularity` is 0.0625, the range of supported line widths **must** be at least [1.0, 7.9375].

8

The minimum `maxDescriptorSet*` limit is  $n$  times the corresponding *specification* minimum `maxPerStageDescriptor*` limit, where  $n$  is the number of shader stages supported by the

`VkPhysicalDevice`. If all shader stages are supported,  $n = 6$  (vertex, tessellation control, tessellation evaluation, geometry, fragment, compute).

9

The `UpdateAfterBind` descriptor limits **must** each be greater than or equal to the corresponding `non-UpdateAfterBind` limit.

## 36.2. Additional Multisampling Capabilities

In addition to the minimum capabilities described for ([Limits](#)) above, implementations **may** support additional multisampling capabilities specific to a particular sample count.

To query additional sample count specific multisampling capabilities, call:

```
void vkGetPhysicalDeviceMultisamplePropertiesEXT(  
    VkPhysicalDevice                      physicalDevice,  
    VkSampleCountFlagBits                 samples,  
    VkMultisamplePropertiesEXT*           pMultisampleProperties);
```

- `physicalDevice` is the physical device from which to query the additional multisampling capabilities.
- `samples` is the sample count to query the capabilities for.
- `pMultisampleProperties` is a pointer to a `VkMultisamplePropertiesEXT` structure in which information about the additional multisampling capabilities specific to the sample count is returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `samples` **must** be a valid `VkSampleCountFlagBits` value
- `pMultisampleProperties` **must** be a valid pointer to a `VkMultisamplePropertiesEXT` structure

The `VkMultisamplePropertiesEXT` structure is defined as

```
typedef struct VkMultisamplePropertiesEXT {  
    VkStructureType    sType;  
    void*             pNext;  
    VkExtent2D        maxSampleLocationGridSize;  
} VkMultisamplePropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `maxSampleLocationGridSize` is the maximum size of the pixel grid in which sample locations **can**

vary.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT`
- `pNext` **must** be `NULL`

If the sample count for which additional multisampling capabilities are requested using `vkGetPhysicalDeviceMultisamplePropertiesEXT` is set in `VkPhysicalDeviceSampleLocationsPropertiesEXT::sampleLocationSampleCounts` the `width` and `height` members of `VkMultisamplePropertiesEXT::maxSampleLocationGridSize` **must** be greater than or equal to the corresponding members of `VkPhysicalDeviceSampleLocationsPropertiesEXT::maxSampleLocationGridSize`, respectively, otherwise both members **must** be `0`.

# Chapter 37. Formats

Supported buffer and image formats **may** vary across implementations. A minimum set of format features are guaranteed, but others **must** be explicitly queried before use to ensure they are supported by the implementation.

The features for the set of formats ([VkFormat](#)) supported by the implementation are queried individually using the [vkGetPhysicalDeviceFormatProperties](#) command.

## 37.1. Format Definition

The following image formats **can** be passed to, and **may** be returned from Vulkan commands. The memory required to store each format is discussed with that format, and also summarized in the [Representation and Texel Block Size](#) section and the [Compatible formats](#) table.

```
typedef enum VkFormat {
    VK_FORMAT_UNDEFINED = 0,
    VK_FORMAT_R4G4_UNORM_PACK8 = 1,
    VK_FORMAT_R4G4B4A4_UNORM_PACK16 = 2,
    VK_FORMAT_B4G4R4A4_UNORM_PACK16 = 3,
    VK_FORMAT_R5G6B5_UNORM_PACK16 = 4,
    VK_FORMAT_B5G6R5_UNORM_PACK16 = 5,
    VK_FORMAT_R5G5B5A1_UNORM_PACK16 = 6,
    VK_FORMAT_B5G5R5A1_UNORM_PACK16 = 7,
    VK_FORMAT_A1R5G5B5_UNORM_PACK16 = 8,
    VK_FORMAT_R8_UNORM = 9,
    VK_FORMAT_R8_SNORM = 10,
    VK_FORMAT_R8_USCALED = 11,
    VK_FORMAT_R8_SSCALED = 12,
    VK_FORMAT_R8_UINT = 13,
    VK_FORMAT_R8_SINT = 14,
    VK_FORMAT_R8_SRGB = 15,
    VK_FORMAT_R8G8_UNORM = 16,
    VK_FORMAT_R8G8_SNORM = 17,
    VK_FORMAT_R8G8_USCALED = 18,
    VK_FORMAT_R8G8_SSCALED = 19,
    VK_FORMAT_R8G8_UINT = 20,
    VK_FORMAT_R8G8_SINT = 21,
    VK_FORMAT_R8G8_SRGB = 22,
    VK_FORMAT_R8G8B8_UNORM = 23,
    VK_FORMAT_R8G8B8_SNORM = 24,
    VK_FORMAT_R8G8B8_USCALED = 25,
    VK_FORMAT_R8G8B8_SSCALED = 26,
    VK_FORMAT_R8G8B8_UINT = 27,
    VK_FORMAT_R8G8B8_SINT = 28,
    VK_FORMAT_R8G8B8_SRGB = 29,
    VK_FORMAT_B8G8R8_UNORM = 30,
    VK_FORMAT_B8G8R8_SNORM = 31,
    VK_FORMAT_B8G8R8_USCALED = 32,
```

```
VK_FORMAT_B8G8R8_SSACLED = 33,  
VK_FORMAT_B8G8R8_UINT = 34,  
VK_FORMAT_B8G8R8_SINT = 35,  
VK_FORMAT_B8G8R8_SRGB = 36,  
VK_FORMAT_R8G8B8A8_UNORM = 37,  
VK_FORMAT_R8G8B8A8_SNORM = 38,  
VK_FORMAT_R8G8B8A8_USCALED = 39,  
VK_FORMAT_R8G8B8A8_SSACLED = 40,  
VK_FORMAT_R8G8B8A8_UINT = 41,  
VK_FORMAT_R8G8B8A8_SINT = 42,  
VK_FORMAT_R8G8B8A8_SRGB = 43,  
VK_FORMAT_B8G8R8A8_UNORM = 44,  
VK_FORMAT_B8G8R8A8_SNORM = 45,  
VK_FORMAT_B8G8R8A8_USCALED = 46,  
VK_FORMAT_B8G8R8A8_SSACLED = 47,  
VK_FORMAT_B8G8R8A8_UINT = 48,  
VK_FORMAT_B8G8R8A8_SINT = 49,  
VK_FORMAT_B8G8R8A8_SRGB = 50,  
VK_FORMAT_A8B8G8R8_UNORM_PACK32 = 51,  
VK_FORMAT_A8B8G8R8_SNORM_PACK32 = 52,  
VK_FORMAT_A8B8G8R8_USCALED_PACK32 = 53,  
VK_FORMAT_A8B8G8R8_SSACLED_PACK32 = 54,  
VK_FORMAT_A8B8G8R8_UINT_PACK32 = 55,  
VK_FORMAT_A8B8G8R8_SINT_PACK32 = 56,  
VK_FORMAT_A8B8G8R8_SRGB_PACK32 = 57,  
VK_FORMAT_A2R10G10B10_UNORM_PACK32 = 58,  
VK_FORMAT_A2R10G10B10_SNORM_PACK32 = 59,  
VK_FORMAT_A2R10G10B10_USCALED_PACK32 = 60,  
VK_FORMAT_A2R10G10B10_SSACLED_PACK32 = 61,  
VK_FORMAT_A2R10G10B10_UINT_PACK32 = 62,  
VK_FORMAT_A2R10G10B10_SINT_PACK32 = 63,  
VK_FORMAT_A2B10G10R10_UNORM_PACK32 = 64,  
VK_FORMAT_A2B10G10R10_SNORM_PACK32 = 65,  
VK_FORMAT_A2B10G10R10_USCALED_PACK32 = 66,  
VK_FORMAT_A2B10G10R10_SSACLED_PACK32 = 67,  
VK_FORMAT_A2B10G10R10_UINT_PACK32 = 68,  
VK_FORMAT_A2B10G10R10_SINT_PACK32 = 69,  
VK_FORMAT_R16_UNORM = 70,  
VK_FORMAT_R16_SNORM = 71,  
VK_FORMAT_R16_USCALED = 72,  
VK_FORMAT_R16_SSACLED = 73,  
VK_FORMAT_R16_UINT = 74,  
VK_FORMAT_R16_SINT = 75,  
VK_FORMAT_R16_SFLOAT = 76,  
VK_FORMAT_R16G16_UNORM = 77,  
VK_FORMAT_R16G16_SNORM = 78,  
VK_FORMAT_R16G16_USCALED = 79,  
VK_FORMAT_R16G16_SSACLED = 80,  
VK_FORMAT_R16G16_UINT = 81,  
VK_FORMAT_R16G16_SINT = 82,  
VK_FORMAT_R16G16_SFLOAT = 83,
```

```
VK_FORMAT_R16G16B16_UNORM = 84,  
VK_FORMAT_R16G16B16_SNORM = 85,  
VK_FORMAT_R16G16B16_USCALED = 86,  
VK_FORMAT_R16G16B16_SSACLED = 87,  
VK_FORMAT_R16G16B16_UINT = 88,  
VK_FORMAT_R16G16B16_SINT = 89,  
VK_FORMAT_R16G16B16_SFLOAT = 90,  
VK_FORMAT_R16G16B16A16_UNORM = 91,  
VK_FORMAT_R16G16B16A16_SNORM = 92,  
VK_FORMAT_R16G16B16A16_USCALED = 93,  
VK_FORMAT_R16G16B16A16_SSACLED = 94,  
VK_FORMAT_R16G16B16A16_UINT = 95,  
VK_FORMAT_R16G16B16A16_SINT = 96,  
VK_FORMAT_R16G16B16A16_SFLOAT = 97,  
VK_FORMAT_R32_UINT = 98,  
VK_FORMAT_R32_SINT = 99,  
VK_FORMAT_R32_SFLOAT = 100,  
VK_FORMAT_R32G32_UINT = 101,  
VK_FORMAT_R32G32_SINT = 102,  
VK_FORMAT_R32G32_SFLOAT = 103,  
VK_FORMAT_R32G32B32_UINT = 104,  
VK_FORMAT_R32G32B32_SINT = 105,  
VK_FORMAT_R32G32B32_SFLOAT = 106,  
VK_FORMAT_R32G32B32A32_UINT = 107,  
VK_FORMAT_R32G32B32A32_SINT = 108,  
VK_FORMAT_R32G32B32A32_SFLOAT = 109,  
VK_FORMAT_R64_UINT = 110,  
VK_FORMAT_R64_SINT = 111,  
VK_FORMAT_R64_SFLOAT = 112,  
VK_FORMAT_R64G64_UINT = 113,  
VK_FORMAT_R64G64_SINT = 114,  
VK_FORMAT_R64G64_SFLOAT = 115,  
VK_FORMAT_R64G64B64_UINT = 116,  
VK_FORMAT_R64G64B64_SINT = 117,  
VK_FORMAT_R64G64B64_SFLOAT = 118,  
VK_FORMAT_R64G64B64A64_UINT = 119,  
VK_FORMAT_R64G64B64A64_SINT = 120,  
VK_FORMAT_R64G64B64A64_SFLOAT = 121,  
VK_FORMAT_B10G11R11_UFLOAT_PACK32 = 122,  
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 = 123,  
VK_FORMAT_D16_UNORM = 124,  
VK_FORMAT_X8_D24_UNORM_PACK32 = 125,  
VK_FORMAT_D32_SFLOAT = 126,  
VK_FORMAT_S8_UINT = 127,  
VK_FORMAT_D16_UNORM_S8_UINT = 128,  
VK_FORMAT_D24_UNORM_S8_UINT = 129,  
VK_FORMAT_D32_SFLOAT_S8_UINT = 130,  
VK_FORMAT_BC1_RGB_UNORM_BLOCK = 131,  
VK_FORMAT_BC1_RGB_SRGB_BLOCK = 132,  
VK_FORMAT_BC1_RGBA_UNORM_BLOCK = 133,  
VK_FORMAT_BC1_RGBA_SRGB_BLOCK = 134,
```

```
VK_FORMAT_BC2_UNORM_BLOCK = 135,  
VK_FORMAT_BC2_SRGB_BLOCK = 136,  
VK_FORMAT_BC3_UNORM_BLOCK = 137,  
VK_FORMAT_BC3_SRGB_BLOCK = 138,  
VK_FORMAT_BC4_UNORM_BLOCK = 139,  
VK_FORMAT_BC4_SNORM_BLOCK = 140,  
VK_FORMAT_BC5_UNORM_BLOCK = 141,  
VK_FORMAT_BC5_SNORM_BLOCK = 142,  
VK_FORMAT_BC6H_UFLOAT_BLOCK = 143,  
VK_FORMAT_BC6H_SFLOAT_BLOCK = 144,  
VK_FORMAT_BC7_UNORM_BLOCK = 145,  
VK_FORMAT_BC7_SRGB_BLOCK = 146,  
VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK = 147,  
VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK = 148,  
VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK = 149,  
VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK = 150,  
VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK = 151,  
VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK = 152,  
VK_FORMAT_EAC_R11_UNORM_BLOCK = 153,  
VK_FORMAT_EAC_R11_SNORM_BLOCK = 154,  
VK_FORMAT_EAC_R11G11_UNORM_BLOCK = 155,  
VK_FORMAT_EAC_R11G11_SNORM_BLOCK = 156,  
VK_FORMAT_ASTC_4x4_UNORM_BLOCK = 157,  
VK_FORMAT_ASTC_4x4_SRGB_BLOCK = 158,  
VK_FORMAT_ASTC_5x4_UNORM_BLOCK = 159,  
VK_FORMAT_ASTC_5x4_SRGB_BLOCK = 160,  
VK_FORMAT_ASTC_5x5_UNORM_BLOCK = 161,  
VK_FORMAT_ASTC_5x5_SRGB_BLOCK = 162,  
VK_FORMAT_ASTC_6x5_UNORM_BLOCK = 163,  
VK_FORMAT_ASTC_6x5_SRGB_BLOCK = 164,  
VK_FORMAT_ASTC_6x6_UNORM_BLOCK = 165,  
VK_FORMAT_ASTC_6x6_SRGB_BLOCK = 166,  
VK_FORMAT_ASTC_8x5_UNORM_BLOCK = 167,  
VK_FORMAT_ASTC_8x5_SRGB_BLOCK = 168,  
VK_FORMAT_ASTC_8x6_UNORM_BLOCK = 169,  
VK_FORMAT_ASTC_8x6_SRGB_BLOCK = 170,  
VK_FORMAT_ASTC_8x8_UNORM_BLOCK = 171,  
VK_FORMAT_ASTC_8x8_SRGB_BLOCK = 172,  
VK_FORMAT_ASTC_10x5_UNORM_BLOCK = 173,  
VK_FORMAT_ASTC_10x5_SRGB_BLOCK = 174,  
VK_FORMAT_ASTC_10x6_UNORM_BLOCK = 175,  
VK_FORMAT_ASTC_10x6_SRGB_BLOCK = 176,  
VK_FORMAT_ASTC_10x8_UNORM_BLOCK = 177,  
VK_FORMAT_ASTC_10x8_SRGB_BLOCK = 178,  
VK_FORMAT_ASTC_10x10_UNORM_BLOCK = 179,  
VK_FORMAT_ASTC_10x10_SRGB_BLOCK = 180,  
VK_FORMAT_ASTC_12x10_UNORM_BLOCK = 181,  
VK_FORMAT_ASTC_12x10_SRGB_BLOCK = 182,  
VK_FORMAT_ASTC_12x12_UNORM_BLOCK = 183,  
VK_FORMAT_ASTC_12x12_SRGB_BLOCK = 184,  
VK_FORMAT_G8B8G8R8_422_UNORM = 1000156000,
```

```
VK_FORMAT_B8G8R8G8_422_UNORM = 1000156001,
VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM = 1000156002,
VK_FORMAT_G8_B8R8_2PLANE_420_UNORM = 1000156003,
VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM = 1000156004,
VK_FORMAT_G8_B8R8_2PLANE_422_UNORM = 1000156005,
VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM = 1000156006,
VK_FORMAT_R10X6_UNORM_PACK16 = 1000156007,
VK_FORMAT_R10X6G10X6_UNORM_2PACK16 = 1000156008,
VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16 = 1000156009,
VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16 = 1000156010,
VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16 = 1000156011,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16 = 1000156012,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16 = 1000156013,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16 = 1000156014,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16 = 1000156015,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16 = 1000156016,
VK_FORMAT_R12X4_UNORM_PACK16 = 1000156017,
VK_FORMAT_R12X4G12X4_UNORM_2PACK16 = 1000156018,
VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16 = 1000156019,
VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16 = 1000156020,
VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16 = 1000156021,
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16 = 1000156022,
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16 = 1000156023,
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16 = 1000156024,
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16 = 1000156025,
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16 = 1000156026,
VK_FORMAT_G16B16G16R16_422_UNORM = 1000156027,
VK_FORMAT_B16G16R16G16_422_UNORM = 1000156028,
VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM = 1000156029,
VK_FORMAT_G16_B16R16_2PLANE_420_UNORM = 1000156030,
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM = 1000156031,
VK_FORMAT_G16_B16R16_2PLANE_422_UNORM = 1000156032,
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM = 1000156033,
VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG = 1000054000,
VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG = 1000054001,
VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG = 1000054002,
VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG = 1000054003,
VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG = 1000054004,
VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG = 1000054005,
VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG = 1000054006,
VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG = 1000054007,
VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT = 1000066000,
VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT = 1000066001,
VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT = 1000066002,
VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT = 1000066003,
VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT = 1000066004,
VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT = 1000066005,
VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT = 1000066006,
VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT = 1000066007,
VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT = 1000066008,
VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT = 1000066009,
```

```

VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT = 1000066010,
VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT = 1000066011,
VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT = 1000066012,
VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT = 1000066013,
VK_FORMAT_G8B8G8R8_422_UNORM_KHR = VK_FORMAT_G8B8G8R8_422_UNORM,
VK_FORMAT_B8G8R8G8_422_UNORM_KHR = VK_FORMAT_B8G8R8G8_422_UNORM,
VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM_KHR = VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM,
VK_FORMAT_G8_B8R8_2PLANE_420_UNORM_KHR = VK_FORMAT_G8_B8R8_2PLANE_420_UNORM,
VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM_KHR = VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM,
VK_FORMAT_G8_B8R8_2PLANE_422_UNORM_KHR = VK_FORMAT_G8_B8R8_2PLANE_422_UNORM,
VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM_KHR = VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM,
VK_FORMAT_R10X6_UNORM_PACK16_KHR = VK_FORMAT_R10X6_UNORM_PACK16,
VK_FORMAT_R10X6G10X6_UNORM_2PACK16_KHR = VK_FORMAT_R10X6G10X6_UNORM_2PACK16,
VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16_KHR =
VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16,
VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16_KHR =
VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16,
VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16_KHR =
VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16_KHR =
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16_KHR =
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16_KHR =
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16_KHR =
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16_KHR =
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16,
VK_FORMAT_R12X4_UNORM_PACK16_KHR = VK_FORMAT_R12X4_UNORM_PACK16,
VK_FORMAT_R12X4G12X4_UNORM_2PACK16_KHR = VK_FORMAT_R12X4G12X4_UNORM_2PACK16,
VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16_KHR =
VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16,
VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16_KHR =
VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16,
VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16_KHR =
VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16,
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16_KHR =
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16,
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16_KHR =
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16,
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16_KHR =
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16,
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16_KHR =
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16,
VK_FORMAT_G16B16G16R16_422_UNORM_KHR = VK_FORMAT_G16B16G16R16_422_UNORM,
VK_FORMAT_B16G16R16G16_422_UNORM_KHR = VK_FORMAT_B16G16R16G16_422_UNORM,
VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM_KHR =
VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM,

```

```

VK_FORMAT_G16_B16R16_2PLANE_420_UNORM_KHR = VK_FORMAT_G16_B16R16_2PLANE_420_UNORM,
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM_KHR =
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM,
VK_FORMAT_G16_B16R16_2PLANE_422_UNORM_KHR = VK_FORMAT_G16_B16R16_2PLANE_422_UNORM,
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM_KHR =
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM,
VK_FORMAT_MAX_ENUM = 0x7FFFFFFF
} VkFormat;

```

- **VK\_FORMAT\_UNDEFINED** specifies that the format is not specified.
- **VK\_FORMAT\_R4G4\_UNORM\_PACK8** specifies a two-component, 8-bit packed unsigned normalized format that has a 4-bit R component in bits 4..7, and a 4-bit G component in bits 0..3.
- **VK\_FORMAT\_R4G4B4A4\_UNORM\_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit R component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit B component in bits 4..7, and a 4-bit A component in bits 0..3.
- **VK\_FORMAT\_B4G4R4A4\_UNORM\_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit B component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit R component in bits 4..7, and a 4-bit A component in bits 0..3.
- **VK\_FORMAT\_R5G6B5\_UNORM\_PACK16** specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit B component in bits 0..4.
- **VK\_FORMAT\_B5G6R5\_UNORM\_PACK16** specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit R component in bits 0..4.
- **VK\_FORMAT\_R5G5B5A1\_UNORM\_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit B component in bits 1..5, and a 1-bit A component in bit 0.
- **VK\_FORMAT\_B5G5R5A1\_UNORM\_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit R component in bits 1..5, and a 1-bit A component in bit 0.
- **VK\_FORMAT\_A1R5G5B5\_UNORM\_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 1-bit A component in bit 15, a 5-bit R component in bits 10..14, a 5-bit G component in bits 5..9, and a 5-bit B component in bits 0..4.
- **VK\_FORMAT\_R8\_UNORM** specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component.
- **VK\_FORMAT\_R8\_SNORM** specifies a one-component, 8-bit signed normalized format that has a single 8-bit R component.
- **VK\_FORMAT\_R8\_USCALED** specifies a one-component, 8-bit unsigned scaled integer format that has a single 8-bit R component.
- **VK\_FORMAT\_R8\_SSACLED** specifies a one-component, 8-bit signed scaled integer format that has a single 8-bit R component.
- **VK\_FORMAT\_R8\_UINT** specifies a one-component, 8-bit unsigned integer format that has a single 8-bit R component.

- `VK_FORMAT_R8_SINT` specifies a one-component, 8-bit signed integer format that has a single 8-bit R component.
- `VK_FORMAT_R8_SRGB` specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component stored with sRGB nonlinear encoding.
- `VK_FORMAT_R8G8_UNORM` specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- `VK_FORMAT_R8G8_SNORM` specifies a two-component, 16-bit signed normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- `VK_FORMAT_R8G8_USCALED` specifies a two-component, 16-bit unsigned scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- `VK_FORMAT_R8G8_SSACLED` specifies a two-component, 16-bit signed scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- `VK_FORMAT_R8G8_UINT` specifies a two-component, 16-bit unsigned integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- `VK_FORMAT_R8G8_SINT` specifies a two-component, 16-bit signed integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.
- `VK_FORMAT_R8G8_SRGB` specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, and an 8-bit G component stored with sRGB nonlinear encoding in byte 1.
- `VK_FORMAT_R8G8B8_UNORM` specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- `VK_FORMAT_R8G8B8_SNORM` specifies a three-component, 24-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- `VK_FORMAT_R8G8B8_USCALED` specifies a three-component, 24-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- `VK_FORMAT_R8G8B8_SSACLED` specifies a three-component, 24-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- `VK_FORMAT_R8G8B8_UINT` specifies a three-component, 24-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- `VK_FORMAT_R8G8B8_SINT` specifies a three-component, 24-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.
- `VK_FORMAT_R8G8B8_SRGB` specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit B component stored with sRGB nonlinear encoding in byte 2.
- `VK_FORMAT_B8G8R8_UNORM` specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- `VK_FORMAT_B8G8R8_SNORM` specifies a three-component, 24-bit signed normalized format that has

an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

- **VK\_FORMAT\_B8G8R8\_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK\_FORMAT\_B8G8R8\_SSACLED** specifies a three-component, 24-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK\_FORMAT\_B8G8R8\_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK\_FORMAT\_B8G8R8\_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.
- **VK\_FORMAT\_B8G8R8\_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit R component stored with sRGB nonlinear encoding in byte 2.
- **VK\_FORMAT\_R8G8B8A8\_UNORM** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_R8G8B8A8\_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_R8G8B8A8\_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_R8G8B8A8\_SSACLED** specifies a four-component, 32-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_R8G8B8A8\_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_R8G8B8A8\_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_R8G8B8A8\_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit B component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_B8G8R8A8\_UNORM** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_B8G8R8A8\_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

- **VK\_FORMAT\_B8G8R8A8\_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_B8G8R8A8\_SSACLED** specifies a four-component, 32-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_B8G8R8A8\_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_B8G8R8A8\_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_B8G8R8A8\_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit R component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.
- **VK\_FORMAT\_A8B8G8R8\_UNORM\_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.
- **VK\_FORMAT\_A8B8G8R8\_SNORM\_PACK32** specifies a four-component, 32-bit packed signed normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.
- **VK\_FORMAT\_A8B8G8R8\_USCALED\_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.
- **VK\_FORMAT\_A8B8G8R8\_SSACLED\_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.
- **VK\_FORMAT\_A8B8G8R8\_UINT\_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.
- **VK\_FORMAT\_A8B8G8R8\_SINT\_PACK32** specifies a four-component, 32-bit packed signed integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.
- **VK\_FORMAT\_A8B8G8R8\_SRGB\_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component stored with sRGB nonlinear encoding in bits 16..23, an 8-bit G component stored with sRGB nonlinear encoding in bits 8..15, and an 8-bit R component stored with sRGB nonlinear encoding in bits 0..7.
- **VK\_FORMAT\_A2R10G10B10\_UNORM\_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.
- **VK\_FORMAT\_A2R10G10B10\_SNORM\_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits

20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK\_FORMAT\_A2R10G10B10\_USCALED\_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.
- **VK\_FORMAT\_A2R10G10B10\_SSACLED\_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.
- **VK\_FORMAT\_A2R10G10B10\_UINT\_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.
- **VK\_FORMAT\_A2R10G10B10\_SINT\_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.
- **VK\_FORMAT\_A2B10G10R10\_UNORM\_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.
- **VK\_FORMAT\_A2B10G10R10\_SNORM\_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.
- **VK\_FORMAT\_A2B10G10R10\_USCALED\_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.
- **VK\_FORMAT\_A2B10G10R10\_SSACLED\_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.
- **VK\_FORMAT\_A2B10G10R10\_UINT\_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.
- **VK\_FORMAT\_A2B10G10R10\_SINT\_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.
- **VK\_FORMAT\_R16\_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit R component.
- **VK\_FORMAT\_R16\_SNORM** specifies a one-component, 16-bit signed normalized format that has a single 16-bit R component.
- **VK\_FORMAT\_R16\_USCALED** specifies a one-component, 16-bit unsigned scaled integer format that has a single 16-bit R component.
- **VK\_FORMAT\_R16\_SSACLED** specifies a one-component, 16-bit signed scaled integer format that has a single 16-bit R component.
- **VK\_FORMAT\_R16\_UINT** specifies a one-component, 16-bit unsigned integer format that has a single 16-bit R component.
- **VK\_FORMAT\_R16\_SINT** specifies a one-component, 16-bit signed integer format that has a single 16-

bit R component.

- **VK\_FORMAT\_R16\_SFLOAT** specifies a one-component, 16-bit signed floating-point format that has a single 16-bit R component.
- **VK\_FORMAT\_R16G16\_UNORM** specifies a two-component, 32-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK\_FORMAT\_R16G16\_SNORM** specifies a two-component, 32-bit signed normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK\_FORMAT\_R16G16\_USCALED** specifies a two-component, 32-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK\_FORMAT\_R16G16\_SSCALED** specifies a two-component, 32-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK\_FORMAT\_R16G16\_UINT** specifies a two-component, 32-bit unsigned integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK\_FORMAT\_R16G16\_SINT** specifies a two-component, 32-bit signed integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK\_FORMAT\_R16G16\_SFLOAT** specifies a two-component, 32-bit signed floating-point format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK\_FORMAT\_R16G16B16\_UNORM** specifies a three-component, 48-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK\_FORMAT\_R16G16B16\_SNORM** specifies a three-component, 48-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK\_FORMAT\_R16G16B16\_USCALED** specifies a three-component, 48-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK\_FORMAT\_R16G16B16\_SSCALED** specifies a three-component, 48-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK\_FORMAT\_R16G16B16\_UINT** specifies a three-component, 48-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK\_FORMAT\_R16G16B16\_SINT** specifies a three-component, 48-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK\_FORMAT\_R16G16B16\_SFLOAT** specifies a three-component, 48-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK\_FORMAT\_R16G16B16A16\_UNORM** specifies a four-component, 64-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

- **VK\_FORMAT\_R16G16B16A16\_SNORM** specifies a four-component, 64-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.
- **VK\_FORMAT\_R16G16B16A16\_USCALED** specifies a four-component, 64-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.
- **VK\_FORMAT\_R16G16B16A16\_SSACLED** specifies a four-component, 64-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.
- **VK\_FORMAT\_R16G16B16A16\_UINT** specifies a four-component, 64-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.
- **VK\_FORMAT\_R16G16B16A16\_SINT** specifies a four-component, 64-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.
- **VK\_FORMAT\_R16G16B16A16\_SFLOAT** specifies a four-component, 64-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.
- **VK\_FORMAT\_R32\_UINT** specifies a one-component, 32-bit unsigned integer format that has a single 32-bit R component.
- **VK\_FORMAT\_R32\_SINT** specifies a one-component, 32-bit signed integer format that has a single 32-bit R component.
- **VK\_FORMAT\_R32\_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has a single 32-bit R component.
- **VK\_FORMAT\_R32G32\_UINT** specifies a two-component, 64-bit unsigned integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.
- **VK\_FORMAT\_R32G32\_SINT** specifies a two-component, 64-bit signed integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.
- **VK\_FORMAT\_R32G32\_SFLOAT** specifies a two-component, 64-bit signed floating-point format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.
- **VK\_FORMAT\_R32G32B32\_UINT** specifies a three-component, 96-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.
- **VK\_FORMAT\_R32G32B32\_SINT** specifies a three-component, 96-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.
- **VK\_FORMAT\_R32G32B32\_SFLOAT** specifies a three-component, 96-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.
- **VK\_FORMAT\_R32G32B32A32\_UINT** specifies a four-component, 128-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

- `VK_FORMAT_R32G32B32A32_SINT` specifies a four-component, 128-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.
- `VK_FORMAT_R32G32B32A32_SFLOAT` specifies a four-component, 128-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.
- `VK_FORMAT_R64_UINT` specifies a one-component, 64-bit unsigned integer format that has a single 64-bit R component.
- `VK_FORMAT_R64_SINT` specifies a one-component, 64-bit signed integer format that has a single 64-bit R component.
- `VK_FORMAT_R64_SFLOAT` specifies a one-component, 64-bit signed floating-point format that has a single 64-bit R component.
- `VK_FORMAT_R64G64_UINT` specifies a two-component, 128-bit unsigned integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.
- `VK_FORMAT_R64G64_SINT` specifies a two-component, 128-bit signed integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.
- `VK_FORMAT_R64G64_SFLOAT` specifies a two-component, 128-bit signed floating-point format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.
- `VK_FORMAT_R64G64B64_UINT` specifies a three-component, 192-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.
- `VK_FORMAT_R64G64B64_SINT` specifies a three-component, 192-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.
- `VK_FORMAT_R64G64B64_SFLOAT` specifies a three-component, 192-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.
- `VK_FORMAT_R64G64B64A64_UINT` specifies a four-component, 256-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.
- `VK_FORMAT_R64G64B64A64_SINT` specifies a four-component, 256-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.
- `VK_FORMAT_R64G64B64A64_SFLOAT` specifies a four-component, 256-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.
- `VK_FORMAT_B10G11R11_UFLOAT_PACK32` specifies a three-component, 32-bit packed unsigned floating-point format that has a 10-bit B component in bits 22..31, an 11-bit G component in bits 11..21, an 11-bit R component in bits 0..10. See [Unsigned 10-Bit Floating-Point Numbers](#) and [Unsigned 11-Bit Floating-Point Numbers](#).
- `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32` specifies a three-component, 32-bit packed unsigned floating-point format that has a 5-bit shared exponent in bits 27..31, a 9-bit B component mantissa in bits

18..26, a 9-bit G component mantissa in bits 9..17, and a 9-bit R component mantissa in bits 0..8.

- **VK\_FORMAT\_D16\_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit depth component.
- **VK\_FORMAT\_X8\_D24\_UNORM\_PACK32** specifies a two-component, 32-bit format that has 24 unsigned normalized bits in the depth component and, optionally:, 8 bits that are unused.
- **VK\_FORMAT\_D32\_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has 32-bits in the depth component.
- **VK\_FORMAT\_S8\_UINT** specifies a one-component, 8-bit unsigned integer format that has 8-bits in the stencil component.
- **VK\_FORMAT\_D16\_UNORM\_S8\_UINT** specifies a two-component, 24-bit format that has 16 unsigned normalized bits in the depth component and 8 unsigned integer bits in the stencil component.
- **VK\_FORMAT\_D24\_UNORM\_S8\_UINT** specifies a two-component, 32-bit packed format that has 8 unsigned integer bits in the stencil component, and 24 unsigned normalized bits in the depth component.
- **VK\_FORMAT\_D32\_SFLOAT\_S8\_UINT** specifies a two-component format that has 32 signed float bits in the depth component and 8 unsigned integer bits in the stencil component. There are optionally: 24-bits that are unused.
- **VK\_FORMAT\_BC1\_RGB\_UNORM\_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.
- **VK\_FORMAT\_BC1\_RGB\_SRGB\_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.
- **VK\_FORMAT\_BC1\_RGBA\_UNORM\_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.
- **VK\_FORMAT\_BC1\_RGBA\_SRGB\_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.
- **VK\_FORMAT\_BC2\_UNORM\_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.
- **VK\_FORMAT\_BC2\_SRGB\_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding.
- **VK\_FORMAT\_BC3\_UNORM\_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.
- **VK\_FORMAT\_BC3\_SRGB\_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB

nonlinear encoding.

- **VK\_FORMAT\_BC4\_UNORM\_BLOCK** specifies a one-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized red texel data.
- **VK\_FORMAT\_BC4\_SNORM\_BLOCK** specifies a one-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of signed normalized red texel data.
- **VK\_FORMAT\_BC5\_UNORM\_BLOCK** specifies a two-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.
- **VK\_FORMAT\_BC5\_SNORM\_BLOCK** specifies a two-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.
- **VK\_FORMAT\_BC6H\_UFLOAT\_BLOCK** specifies a three-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned floating-point RGB texel data.
- **VK\_FORMAT\_BC6H\_SFLOAT\_BLOCK** specifies a three-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed floating-point RGB texel data.
- **VK\_FORMAT\_BC7\_UNORM\_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.
- **VK\_FORMAT\_BC7\_SRGB\_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- **VK\_FORMAT\_ETC2\_R8G8B8\_UNORM\_BLOCK** specifies a three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.
- **VK\_FORMAT\_ETC2\_R8G8B8\_SRGB\_BLOCK** specifies a three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.
- **VK\_FORMAT\_ETC2\_R8G8B8A1\_UNORM\_BLOCK** specifies a four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.
- **VK\_FORMAT\_ETC2\_R8G8B8A1\_SRGB\_BLOCK** specifies a four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.
- **VK\_FORMAT\_ETC2\_R8G8B8A8\_UNORM\_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.
- **VK\_FORMAT\_ETC2\_R8G8B8A8\_SRGB\_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding applied.

- `VK_FORMAT_EAC_R11_UNORM_BLOCK` specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized red texel data.
- `VK_FORMAT_EAC_R11_SNORM_BLOCK` specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of signed normalized red texel data.
- `VK_FORMAT_EAC_R11G11_UNORM_BLOCK` specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.
- `VK_FORMAT_EAC_R11G11_SNORM_BLOCK` specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of signed normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.
- `VK_FORMAT_ASTC_4x4_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_4x4_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $4 \times 4$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_5x4_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $5 \times 4$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_5x4_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $5 \times 4$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $5 \times 4$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_5x5_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $5 \times 5$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_5x5_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $5 \times 5$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $5 \times 5$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_6x5_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $6 \times 5$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_6x5_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $6 \times 5$  rectangle of unsigned normalized RGBA texel

data with sRGB nonlinear encoding applied to the RGB components.

- `VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $6 \times 5$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_6x6_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $6 \times 6$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_6x6_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $6 \times 6$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $6 \times 6$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_8x5_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an  $8 \times 5$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_8x5_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an  $8 \times 5$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $8 \times 5$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_8x6_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an  $8 \times 6$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_8x6_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an  $8 \times 6$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $8 \times 6$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_8x8_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an  $8 \times 8$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_8x8_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an  $8 \times 8$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $8 \times 8$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_10x5_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 5$  rectangle of unsigned normalized RGBA

texel data.

- `VK_FORMAT_ASTC_10x5_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 5$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 5$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_10x6_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 6$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_10x6_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 6$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 6$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_10x8_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 8$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_10x8_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 8$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 8$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_10x10_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 10$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_10x10_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 10$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $10 \times 10$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_ASTC_12x10_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $12 \times 10$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_12x10_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $12 \times 10$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $12 \times 10$  rectangle of signed floating-point

RGBA texel data.

- `VK_FORMAT_ASTC_12x12_UNORM_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $12 \times 12$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_ASTC_12x12_SRGB_BLOCK` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $12 \times 12$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT` specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a  $12 \times 12$  rectangle of signed floating-point RGBA texel data.
- `VK_FORMAT_G8B8G8R8_422_UNORM` specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit G component for the even  $i$  coordinate in byte 0, an 8-bit B component in byte 1, an 8-bit G component for the odd  $i$  coordinate in byte 2, and an 8-bit R component in byte 3. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a  $2 \times 1$  compressed texel block.
- `VK_FORMAT_B8G8R8G8_422_UNORM` specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit B component in byte 0, an 8-bit G component for the even  $i$  coordinate in byte 1, an 8-bit R component in byte 2, and an 8-bit G component for the odd  $i$  coordinate in byte 3. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a  $2 \times 1$  compressed texel block.
- `VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM` specifies an unsigned normalized *multi-planar format* that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$  and  $|j_G \times 0.5| = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. Images in this format **must** be defined with a width and height that is a multiple of two.
- `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM` specifies an unsigned normalized *multi-planar format* that has an 8-bit G component in plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. The horizontal and vertical dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$  and  $|j_G \times 0.5| = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. Images in this format **must** be defined with a width and height that is a multiple of two.

width and height that is a multiple of two.

- **VK\_FORMAT\_G8\_B8\_R8\_3PLANE\_422\_UNORM** specifies an unsigned normalized *multi-planar format* that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using **VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT** for the G plane, **VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT** for the B plane, and **VK\_IMAGE\_ASPECT\_PLANE\_2\_BIT** for the R plane. Images in this format **must** be defined with a width that is a multiple of two.
- **VK\_FORMAT\_G8\_B8R8\_2PLANE\_422\_UNORM** specifies an unsigned normalized *multi-planar format* that has an 8-bit G component in plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. The horizontal dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using **VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT** for the G plane, and **VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT** for the BR plane. Images in this format **must** be defined with a width that is a multiple of two.
- **VK\_FORMAT\_G8\_B8\_R8\_3PLANE\_444\_UNORM** specifies an unsigned normalized *multi-planar format* that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using **VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT** for the G plane, **VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT** for the B plane, and **VK\_IMAGE\_ASPECT\_PLANE\_2\_BIT** for the R plane.
- **VK\_FORMAT\_R10X6\_UNORM\_PACK16** specifies a one-component, 16-bit unsigned normalized format that has a single 10-bit R component in the top 10 bits of a 16-bit word, with the bottom 6 bits set to 0.
- **VK\_FORMAT\_R10X6G10X6\_UNORM\_2PACK16** specifies a two-component, 32-bit unsigned normalized format that has a 10-bit R component in the top 10 bits of the word in bytes 0..1, and a 10-bit G component in the top 10 bits of the word in bytes 2..3, with the bottom 6 bits of each word set to 0.
- **VK\_FORMAT\_R10X6G10X6B10X6A10X6\_UNORM\_4PACK16** specifies a four-component, 64-bit unsigned normalized format that has a 10-bit R component in the top 10 bits of the word in bytes 0..1, a 10-bit G component in the top 10 bits of the word in bytes 2..3, a 10-bit B component in the top 10 bits of the word in bytes 4..5, and a 10-bit A component in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word set to 0.
- **VK\_FORMAT\_G10X6B10X6G10X6R10X6\_422\_UNORM\_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit G component for the even  $i$  coordinate in the top 10 bits of the word in bytes 0..1, a 10-bit B component in the top 10 bits of the word in bytes 2..3, a 10-bit G component for the odd  $i$  coordinate in the top 10 bits of the word in bytes 4..5, and a 10-bit R component in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word set to 0. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a

compressed format with a  $2 \times 1$  compressed texel block.

- `VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16` specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit B component in the top 10 bits of the word in bytes 0..1, a 10-bit G component for the even  $i$  coordinate in the top 10 bits of the word in bytes 2..3, a 10-bit R component in the top 10 bits of the word in bytes 4..5, and a 10-bit G component for the odd  $i$  coordinate in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word set to 0. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a  $2 \times 1$  compressed texel block.
- `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16` specifies an unsigned normalized *multi-planar format* that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word set to 0. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$  and  $|j_G \times 0.5| = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. Images in this format **must** be defined with a width and height that is a multiple of two.
- `VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16` specifies an unsigned normalized *multi-planar format* that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of the word in bytes 0..1, and a 10-bit R component in the top 10 bits of the word in bytes 2..3, the bottom 6 bits of each word set to 0. The horizontal and vertical dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$  and  $|j_G \times 0.5| = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. Images in this format **must** be defined with a width and height that is a multiple of two.
- `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16` specifies an unsigned normalized *multi-planar format* that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word set to 0. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. Images in this format **must** be defined with a width that is a multiple of two.
- `VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16` specifies an unsigned normalized *multi-planar format* that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of

the word in bytes 0..1, and a 10-bit R component in the top 10 bits of the word in bytes 2..3, the bottom 6 bits of each word set to 0. The horizontal dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. Images in this format **must** be defined with a width that is a multiple of two.

- `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16` specifies an unsigned normalized *multi-planar format* that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word set to 0. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.
- `VK_FORMAT_R12X4_UNORM_PACK16` specifies a one-component, 16-bit unsigned normalized format that has a single 12-bit R component in the top 12 bits of a 16-bit word, with the bottom 4 bits set to 0.
- `VK_FORMAT_R12X4G12X4_UNORM_2PACK16` specifies a two-component, 32-bit unsigned normalized format that has a 12-bit R component in the top 12 bits of the word in bytes 0..1, and a 12-bit G component in the top 12 bits of the word in bytes 2..3, with the bottom 4 bits of each word set to 0.
- `VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16` specifies a four-component, 64-bit unsigned normalized format that has a 12-bit R component in the top 12 bits of the word in bytes 0..1, a 12-bit G component in the top 12 bits of the word in bytes 2..3, a 12-bit B component in the top 12 bits of the word in bytes 4..5, and a 12-bit A component in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word set to 0.
- `VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16` specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 12-bit G component for the even  $i$  coordinate in the top 12 bits of the word in bytes 0..1, a 12-bit B component in the top 12 bits of the word in bytes 2..3, a 12-bit G component for the odd  $i$  coordinate in the top 12 bits of the word in bytes 4..5, and a 12-bit R component in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word set to 0. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a  $2 \times 1$  compressed texel block.
- `VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16` specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 12-bit B component in the top 12 bits of the word in bytes 0..1, a 12-bit G component for the even  $i$  coordinate in the top 12 bits of the word in bytes 2..3, a 12-bit R component in the top 12 bits of the word in bytes 4..5, and a 12-bit G

component for the odd  $i$  coordinate in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word set to 0. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a  $2 \times 1$  compressed texel block.

- **VK\_FORMAT\_G12X4\_B12X4\_R12X4\_3PLANE\_420\_UNORM\_3PACK16** specifies an unsigned normalized *multi-planar format* that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word set to 0. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$  and  $|j_G \times 0.5| = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via [vkGetImageSubresourceLayout](#), using **VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT** for the G plane, **VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT** for the B plane, and **VK\_IMAGE\_ASPECT\_PLANE\_2\_BIT** for the R plane. Images in this format **must** be defined with a width and height that is a multiple of two.
- **VK\_FORMAT\_G12X4\_B12X4R12X4\_2PLANE\_420\_UNORM\_3PACK16** specifies an unsigned normalized *multi-planar format* that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, the bottom 4 bits of each word set to 0. The horizontal and vertical dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$  and  $|j_G \times 0.5| = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via [vkGetImageSubresourceLayout](#), using **VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT** for the G plane, and **VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT** for the BR plane. Images in this format **must** be defined with a width and height that is a multiple of two.
- **VK\_FORMAT\_G12X4\_B12X4\_R12X4\_3PLANE\_422\_UNORM\_3PACK16** specifies an unsigned normalized *multi-planar format* that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word set to 0. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via [vkGetImageSubresourceLayout](#), using **VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT** for the G plane, **VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT** for the B plane, and **VK\_IMAGE\_ASPECT\_PLANE\_2\_BIT** for the R plane. Images in this format **must** be defined with a width that is a multiple of two.
- **VK\_FORMAT\_G12X4\_B12X4R12X4\_2PLANE\_422\_UNORM\_3PACK16** specifies an unsigned normalized *multi-planar format* that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, the bottom 4 bits of each word set to 0. The horizontal dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via [vkGetImageSubresourceLayout](#), using **VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT** for the G plane, and **VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT** for the BR plane. Images in this format **must** be defined with a width that is a multiple of two.
- **VK\_FORMAT\_G12X4\_B12X4\_R12X4\_3PLANE\_444\_UNORM\_3PACK16** specifies an unsigned normalized *multi-*

*planar format* that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word set to 0. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- `VK_FORMAT_G16B16G16R16_422_UNORM` specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 16-bit G component for the even  $i$  coordinate in the word in bytes 0..1, a 16-bit B component in the word in bytes 2..3, a 16-bit G component for the odd  $i$  coordinate in the word in bytes 4..5, and a 16-bit R component in the word in bytes 6..7. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a  $2 \times 1$  compressed texel block.
- `VK_FORMAT_B16G16R16G16_422_UNORM` specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a  $2 \times 1$  rectangle of unsigned normalized RGB texel data. One G value is present at each  $i$  coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 16-bit B component in the word in bytes 0..1, a 16-bit G component for the even  $i$  coordinate in the word in bytes 2..3, a 16-bit R component in the word in bytes 4..5, and a 16-bit G component for the odd  $i$  coordinate in the word in bytes 6..7. Images in this format **must** be defined with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a  $2 \times 1$  compressed texel block.
- `VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM` specifies an unsigned normalized *multi-planar format* that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which  $\lfloor i_G \times 0.5 \rfloor = i_B = i_R$  and  $\lfloor j_G \times 0.5 \rfloor = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. Images in this format **must** be defined with a width and height that is a multiple of two.
- `VK_FORMAT_G16_B16R16_2PLANE_420_UNORM` specifies an unsigned normalized *multi-planar format* that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. The horizontal and vertical dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $\lfloor i_G \times 0.5 \rfloor = i_B = i_R$  and  $\lfloor j_G \times 0.5 \rfloor = j_B = j_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. Images in this format **must** be defined with a width and height that is a multiple of two.
- `VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM` specifies an unsigned normalized *multi-planar format* that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-

bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. Images in this format **must** be defined with a width that is a multiple of two.

- `VK_FORMAT_G16_B16R16_2PLANE_422_UNORM` specifies an unsigned normalized *multi-planar format* that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. The horizontal dimensions of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which  $|i_G \times 0.5| = i_B = i_R$ . The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. Images in this format **must** be defined with a width that is a multiple of two.
- `VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM` specifies an unsigned normalized *multi-planar format* that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.
- `VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an  $8 \times 4$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an  $8 \times 4$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data.
- `VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an  $8 \times 4$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.
- `VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an  $8 \times 4$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- `VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG` specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a  $4 \times 4$  rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

### 37.1.1. Compatible formats of planes of multi-planar formats

Individual planes of multi-planar formats are *compatible* with single-plane formats if they occupy the same number of bits per texel block. In the following table, individual planes of a *multi-planar* format are compatible with the format listed against the relevant plane index for that multi-planar format.

*Table 47. Plane Format Compatibility Table*

Plane	Compatible format for plane	Width relative to the width $w$ of the plane with the largest dimensions	Height relative to the height $h$ of the plane with the largest dimensions
<b><code>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</code></b>			
0	<code>VK_FORMAT_R8_UNORM</code>	$w$	$h$
1	<code>VK_FORMAT_R8_UNORM</code>	$w/2$	$h/2$
2	<code>VK_FORMAT_R8_UNORM</code>	$w/2$	$h/2$
<b><code>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</code></b>			
0	<code>VK_FORMAT_R8_UNORM</code>	$w$	$h$
1	<code>VK_FORMAT_R8G8_UNORM</code>	$w/2$	$h/2$
<b><code>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</code></b>			
0	<code>VK_FORMAT_R8_UNORM</code>	$w$	$h$
1	<code>VK_FORMAT_R8_UNORM</code>	$w/2$	$h$
2	<code>VK_FORMAT_R8_UNORM</code>	$w/2$	$h$
<b><code>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</code></b>			
0	<code>VK_FORMAT_R8_UNORM</code>	$w$	$h$
1	<code>VK_FORMAT_R8G8_UNORM</code>	$w/2$	$h$
<b><code>VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM</code></b>			
0	<code>VK_FORMAT_R8_UNORM</code>	$w$	$h$
1	<code>VK_FORMAT_R8_UNORM</code>	$w$	$h$
2	<code>VK_FORMAT_R8_UNORM</code>	$w$	$h$
<b><code>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</code></b>			
0	<code>VK_FORMAT_R10X6_UNORM_PACK16</code>	$w$	$h$
1	<code>VK_FORMAT_R10X6_UNORM_PACK16</code>	$w/2$	$h/2$
2	<code>VK_FORMAT_R10X6_UNORM_PACK16</code>	$w/2$	$h/2$
<b><code>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</code></b>			
0	<code>VK_FORMAT_R10X6_UNORM_PACK16</code>	$w$	$h$
1	<code>VK_FORMAT_R10X6G10X6_UNORM_2PACK16</code>	$w/2$	$h/2$
<b><code>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</code></b>			

<b>Plane</b>	<b>Compatible format for plane</b>	<b>Width relative to the width <math>w</math> of the plane with the largest dimensions</b>	<b>Height relative to the height <math>h</math> of the plane with the largest dimensions</b>
0	VK_FORMAT_R10X6_UNORM_PACK16	w	h
1	VK_FORMAT_R10X6_UNORM_PACK16	w/2	h
2	VK_FORMAT_R10X6_UNORM_PACK16	w/2	h
<b>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</b>			
0	VK_FORMAT_R10X6_UNORM_PACK16	w	h
1	VK_FORMAT_R10X6G10X6_UNORM_2PACK16	w/2	h
<b>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</b>			
0	VK_FORMAT_R10X6_UNORM_PACK16	w	h
1	VK_FORMAT_R10X6_UNORM_PACK16	w	h
2	VK_FORMAT_R10X6_UNORM_PACK16	w	h
<b>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</b>			
0	VK_FORMAT_R12X4_UNORM_PACK16	w	h
1	VK_FORMAT_R12X4_UNORM_PACK16	w/2	h/2
2	VK_FORMAT_R12X4_UNORM_PACK16	w/2	h/2
<b>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</b>			
0	VK_FORMAT_R12X4_UNORM_PACK16	w	h
1	VK_FORMAT_R12X4G12X4_UNORM_2PACK16	w/2	h/2
<b>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</b>			
0	VK_FORMAT_R12X4_UNORM_PACK16	w	h
1	VK_FORMAT_R12X4_UNORM_PACK16	w/2	h
2	VK_FORMAT_R12X4_UNORM_PACK16	w/2	h
<b>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</b>			
0	VK_FORMAT_R12X4_UNORM_PACK16	w	h
1	VK_FORMAT_R12X4G12X4_UNORM_2PACK16	w/2	h
<b>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</b>			
0	VK_FORMAT_R12X4_UNORM_PACK16	w	h
1	VK_FORMAT_R12X4_UNORM_PACK16	w	h
2	VK_FORMAT_R12X4_UNORM_PACK16	w	h
<b>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</b>			
0	VK_FORMAT_R16_UNORM	w	h
1	VK_FORMAT_R16_UNORM	w/2	h/2
2	VK_FORMAT_R16_UNORM	w/2	h/2
<b>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</b>			
0	VK_FORMAT_R16_UNORM	w	h
1	VK_FORMAT_R16G16_UNORM	w/2	h/2

Plane	Compatible format for plane	Width relative to the width $w$ of the plane with the largest dimensions	Height relative to the height $h$ of the plane with the largest dimensions
<b>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</b>			
0	VK_FORMAT_R16_UNORM	w	h
1	VK_FORMAT_R16_UNORM	w/2	h
2	VK_FORMAT_R16_UNORM	w/2	h
<b>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</b>			
0	VK_FORMAT_R16_UNORM	w	h
1	VK_FORMAT_R16G16_UNORM	w/2	h
<b>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</b>			
0	VK_FORMAT_R16_UNORM	w	h
1	VK_FORMAT_R16_UNORM	w	h
2	VK_FORMAT_R16_UNORM	w	h

### 37.1.2. Packed Formats

For the purposes of address alignment when accessing buffer memory containing vertex attribute or texel data, the following formats are considered *packed* - whole texels or attributes are stored in bitfields of a single 8-, 16-, or 32-bit fundamental data type.

- Packed into 8-bit data types:
  - VK\_FORMAT\_R4G4\_UNORM\_PACK8
- Packed into 16-bit data types:
  - VK\_FORMAT\_R4G4B4A4\_UNORM\_PACK16
  - VK\_FORMAT\_B4G4R4A4\_UNORM\_PACK16
  - VK\_FORMAT\_R5G6B5\_UNORM\_PACK16
  - VK\_FORMAT\_B5G6R5\_UNORM\_PACK16
  - VK\_FORMAT\_R5G5B5A1\_UNORM\_PACK16
  - VK\_FORMAT\_B5G5R5A1\_UNORM\_PACK16
  - VK\_FORMAT\_A1R5G5B5\_UNORM\_PACK16
- Packed into 32-bit data types:
  - VK\_FORMAT\_A8B8G8R8\_UNORM\_PACK32
  - VK\_FORMAT\_A8B8G8R8\_SNORM\_PACK32
  - VK\_FORMAT\_A8B8G8R8\_USCALED\_PACK32
  - VK\_FORMAT\_A8B8G8R8\_SSACLED\_PACK32
  - VK\_FORMAT\_A8B8G8R8\_UINT\_PACK32
  - VK\_FORMAT\_A8B8G8R8\_SINT\_PACK32
  - VK\_FORMAT\_A8B8G8R8\_SRGB\_PACK32
  - VK\_FORMAT\_A2R10G10B10\_UNORM\_PACK32
  - VK\_FORMAT\_A2R10G10B10\_SNORM\_PACK32

- VK\_FORMAT\_A2R10G10B10\_USCALED\_PACK32
- VK\_FORMAT\_A2R10G10B10\_SSACLED\_PACK32
- VK\_FORMAT\_A2R10G10B10\_UINT\_PACK32
- VK\_FORMAT\_A2R10G10B10\_SINT\_PACK32
- VK\_FORMAT\_A2B10G10R10\_UNORM\_PACK32
- VK\_FORMAT\_A2B10G10R10\_SNORM\_PACK32
- VK\_FORMAT\_A2B10G10R10\_USCALED\_PACK32
- VK\_FORMAT\_A2B10G10R10\_SSACLED\_PACK32
- VK\_FORMAT\_A2B10G10R10\_UINT\_PACK32
- VK\_FORMAT\_A2B10G10R10\_SINT\_PACK32
- VK\_FORMAT\_B10G11R11\_UFLOAT\_PACK32
- VK\_FORMAT\_E5B9G9R9\_UFLOAT\_PACK32
- VK\_FORMAT\_X8\_D24\_UNORM\_PACK32

### 37.1.3. Identification of Formats

A “format” is represented by a single enum value. The name of a format is usually built up by using the following pattern:

```
VK_FORMAT_{component-format|compression-scheme}_{numeric-format}
```

The component-format indicates either the size of the R, G, B, and A components (if they are present) in the case of a color format, or the size of the depth (D) and stencil (S) components (if they are present) in the case of a depth/stencil format (see below). An X indicates a component that is unused, but **may** be present for padding.

Table 48. Interpretation of Numeric Format

Numeric format	SPIR-V Sampled Type	Description
UNORM	OpTypeFloat	The components are unsigned normalized values in the range [0,1]
SNORM	OpTypeFloat	The components are signed normalized values in the range [-1,1]
USCALED	OpTypeFloat	The components are unsigned integer values that get converted to floating-point in the range [0, $2^{n-1}$ -1]
SSCALED	OpTypeFloat	The components are signed integer values that get converted to floating-point in the range [- $2^{n-1}$ , $2^{n-1}$ -1]
UINT	OpTypeInt	The components are unsigned integer values in the range [0, $2^{n-1}$ -1]
SINT	OpTypeInt	The components are signed integer values in the range [- $2^{n-1}$ , $2^{n-1}$ -1]
UFLOAT	OpTypeFloat	The components are unsigned floating-point numbers (used by packed, shared exponent, and some compressed formats)
SFLOAT	OpTypeFloat	The components are signed floating-point numbers
SRGB	OpTypeFloat	The R, G, and B components are unsigned normalized values that represent values using sRGB nonlinear encoding, while the A component (if one exists) is a regular unsigned normalized value

The suffix `_PACKnn` indicates that the format is packed into an underlying type with nn bits. The suffix `_mPACKnn` is a short-hand that indicates that the format has several components (which may or may not be stored in separate *planes*) that are each packed into an underlying type with nn bits.

The suffix `_BLOCK` indicates that the format is a block-compressed format, with the representation of multiple pixels encoded interdependently within a region.

Table 49. Interpretation of Compression Scheme

Compression scheme	Description
BC	Block Compression. See <a href="#">Block-Compressed Image Formats</a> .
ETC2	Ericsson Texture Compression. See <a href="#">ETC Compressed Image Formats</a> .
EAC	ETC2 Alpha Compression. See <a href="#">ETC Compressed Image Formats</a> .
ASTC	Adaptive Scalable Texture Compression (LDR Profile). See <a href="#">ASTC Compressed Image Formats</a> .

For *multi-planar* images, the components in separate *planes* are separated by underscores, and the number of planes is indicated by the addition of a `_2PLANE` or `_3PLANE` suffix. Similarly, the separate aspects of depth-stencil formats are separated by underscores, although these are not considered separate planes. Formats are suffixed by `_422` to indicate that planes other than the first are reduced in size by a factor of two horizontally or that the R and B values appear at half the horizontal frequency of the G values, `_420` to indicate that planes other than the first are reduced in

size by a factor of two both horizontally and vertically, and [\\_444](#) for consistency to indicate that all three planes of a three-planar image are the same size.

**Note**



No common format has a single plane containing both R and B channels but does not store these channels at reduced horizontal resolution.

### 37.1.4. Representation and Texel Block Size

Color formats **must** be represented in memory in exactly the form indicated by the format's name. This means that promoting one format to another with more bits per component and/or additional components **must** not occur for color formats. Depth/stencil formats have more relaxed requirements as discussed [below](#).

Each format has a *texel block size*, the number of bytes used to store one *texel block* (a single addressable element of an uncompressed image, or a single compressed block of a compressed image). The texel block size for each format is shown in the [Compatible formats](#) table.

The representation of non-packed formats is that the first component specified in the name of the format is in the lowest memory addresses and the last component specified is in the highest memory addresses. See [Byte mappings for non-packed/compressed color formats](#). The in-memory ordering of bytes within a component is determined by the host endianness.

*Table 50. Byte mappings for non-packed/compressed color formats*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	← Byte
R																VK_FORMAT_R8_*
R	G															VK_FORMAT_R8G8_*
R	G	B														VK_FORMAT_R8G8B8_*
B	G	R														VK_FORMAT_B8G8R8_*
R	G	B	A													VK_FORMAT_R8G8B8A8_*
B	G	R	A													VK_FORMAT_B8G8R8A8_*
G <sub>0</sub>	B	G <sub>1</sub>	R													VK_FORMAT_G8B8G8R8_422_UNORM
B	G <sub>0</sub>	R	G <sub>1</sub>													VK_FORMAT_B8G8R8G8_422_UNORM
R																VK_FORMAT_R16_*
R	G															VK_FORMAT_R16G16_*
R	G	B														VK_FORMAT_R16G16B16_*
R	G	B	A													VK_FORMAT_R16G16B16A16_*
G <sub>0</sub>	B	G <sub>1</sub>	R													VK_FORMAT_G10X6B10X6G10X6R10X6_4PACK16_422_UNORM VK_FORMAT_G12X4B12X4G12X4R12X4_4PACK16_422_UNORM VK_FORMAT_G16B16G16R16_UNORM
B	G <sub>0</sub>	R	G <sub>1</sub>													VK_FORMAT_B10X6G10X6R10X6G10X6_4PACK16_422_UNORM VK_FORMAT_B12X4G12X4R12X4G12X4_4PACK16_422_UNORM VK_FORMAT_B16G16R16G16_422_UNORM
R																VK_FORMAT_R32_*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	← Byte
R		G														VK_FORMAT_R32G32_*
R		G		B												VK_FORMAT_R32G32B32_*
R		G		B			A									VK_FORMAT_R32G32B32A32_*
R																VK_FORMAT_R64_*
R				G												VK_FORMAT_R64G64_*
<b>VK_FORMAT_R64G64B64_*</b> as <b>VK_FORMAT_R64G64_*</b> but with B in bytes 16-23																
<b>VK_FORMAT_R64G64B64A64_*</b> as <b>VK_FORMAT_R64G64B64_*</b> but with A in bytes 24-31																

Packed formats store multiple components within one underlying type. The bit representation is that the first component specified in the name of the format is in the most-significant bits and the last component specified is in the least-significant bits of the underlying type. The in-memory ordering of bytes comprising the underlying type is determined by the host endianness.

Table 51. Bit mappings for packed 8-bit formats

Bit															
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
<b>VK_FORMAT_R4G4_UNORM_PACK8</b>															
<b>R</b>								<b>G</b>							
3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0

Table 52. Bit mappings for packed 16-bit formats

Bit															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<b>VK_FORMAT_R4G4B4A4_UNORM_PACK16</b>															
<b>R</b>				<b>G</b>				<b>B</b>				<b>A</b>			
3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
<b>VK_FORMAT_B4G4R4A4_UNORM_PACK16</b>															
<b>B</b>				<b>G</b>				<b>R</b>				<b>A</b>			
3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
<b>VK_FORMAT_R5G6B5_UNORM_PACK16</b>															
<b>R</b>					<b>G</b>					<b>B</b>					
4	3	2	1	0	5	4	3	2	1	0	4	3	2	1	0
<b>VK_FORMAT_B5G6R5_UNORM_PACK16</b>															
<b>B</b>					<b>G</b>					<b>R</b>					
4	3	2	1	0	5	4	3	2	1	0	4	3	2	1	0
<b>VK_FORMAT_R5G5B5A1_UNORM_PACK16</b>															
<b>R</b>					<b>G</b>					<b>B</b>					<b>A</b>
4	3	2	1	0	4	3	2	1	0	4	3	2	1	0	0
<b>VK_FORMAT_B5G5R5A1_UNORM_PACK16</b>															
<b>B</b>					<b>G</b>					<b>R</b>					<b>A</b>
4	3	2	1	0	4	3	2	1	0	4	3	2	1	0	0
<b>VK_FORMAT_A1R5G5B5_UNORM_PACK16</b>															

Bit																
A	R						G					B				
0	4	3	2	1	0		4	3	2	1	0	4	3	2	1	0
VK_FORMAT_R10X6_UNORM_PACK16																
R								X								
9	8	7	6	5	4	3	2	1	0	5	4	3	2	1	0	
VK_FORMAT_R12X4_UNORM_PACK16																
R								X								
11	10	9	8	7	6	5	4	3	2	1	0	3	2	1	0	

Table 53. Bit mappings for packed 32-bit formats

Bit																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VK_FORMAT_A8B8G8R8_*_PACK32																															
A						B						G					R														
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
VK_FORMAT_A2R10G10B10_*_PACK32																															
A	R						G						B					R													
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
VK_FORMAT_A2B10G10R10_*_PACK32																															
A	B						G						R					R													
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
VK_FORMAT_B10G11R11_UFLOAT_PACK32																															
B								G								R															
9	8	7	6	5	4	3	2	1	0	10	9	8	7	6	5	4	3	2	1	0	10	9	8	7	6	5	4	3	2	1	0
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32																															
E	B						G						R					R													
4	3	2	1	0	8	7	6	5	4	3	2	1	0	8	7	6	5	4	3	2	1	0	8	7	6	5	4	3	2	1	0
VK_FORMAT_X8_D24_UNORM_PACK32																															
X	D																														
7	6	5	4	3	2	1	0	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

### 37.1.5. Depth/Stencil Formats

Depth/stencil formats are considered opaque and need not be stored in the exact number of bits per texel or component ordering indicated by the format enum. However, implementations **must** not substitute a different depth or stencil precision than that described in the format (e.g. D16 **must** not be implemented as D24 or D32).

### 37.1.6. Format Compatibility Classes

Uncompressed color formats are *compatible* with each other if they occupy the same number of bits per texel block. Compressed color formats are compatible with each other if the only difference between them is the numerical type of the uncompressed pixels (e.g. signed vs. unsigned, or SRGB vs. UNORM encoding). Each depth/stencil format is only compatible with itself. In the [following](#)

table, all the formats in the same row are compatible.

Table 54. Compatible Formats

Class, Texel Block Size, # Texels/Block	Formats
8-bit Block size 1 byte 1 texel/block	<code>VK_FORMAT_R4G4_UNORM_PACK8,</code> <code>VK_FORMAT_R8_UNORM,</code> <code>VK_FORMAT_R8_SNORM,</code> <code>VK_FORMAT_R8_USCALED,</code> <code>VK_FORMAT_R8_SSACLED,</code> <code>VK_FORMAT_R8_UINT,</code> <code>VK_FORMAT_R8_SINT,</code> <code>VK_FORMAT_R8_SRGB</code>
16-bit Block size 2 bytes 1 texel/block	<code>VK_FORMAT_R4G4B4A4_UNORM_PACK16,</code> <code>VK_FORMAT_B4G4R4A4_UNORM_PACK16,</code> <code>VK_FORMAT_R5G6B5_UNORM_PACK16,</code> <code>VK_FORMAT_B5G6R5_UNORM_PACK16,</code> <code>VK_FORMAT_R5G5B5A1_UNORM_PACK16,</code> <code>VK_FORMAT_B5G5R5A1_UNORM_PACK16,</code> <code>VK_FORMAT_A1R5G5B5_UNORM_PACK16,</code> <code>VK_FORMAT_R8G8_UNORM,</code> <code>VK_FORMAT_R8G8_SNORM,</code> <code>VK_FORMAT_R8G8_USCALED,</code> <code>VK_FORMAT_R8G8_SSACLED,</code> <code>VK_FORMAT_R8G8_UINT,</code> <code>VK_FORMAT_R8G8_SINT,</code> <code>VK_FORMAT_R8G8_SRGB,</code> <code>VK_FORMAT_R16_UNORM,</code> <code>VK_FORMAT_R16_SNORM,</code> <code>VK_FORMAT_R16_USCALED,</code> <code>VK_FORMAT_R16_SSACLED,</code> <code>VK_FORMAT_R16_UINT,</code> <code>VK_FORMAT_R16_SINT,</code> <code>VK_FORMAT_R16_SFLOAT,</code> <code>VK_FORMAT_R10X6_UNORM_PACK16,</code> <code>VK_FORMAT_R12X4_UNORM_PACK16</code>
24-bit Block size 3 bytes 1 texel/block	<code>VK_FORMAT_R8G8B8_UNORM,</code> <code>VK_FORMAT_R8G8B8_SNORM,</code> <code>VK_FORMAT_R8G8B8_USCALED,</code> <code>VK_FORMAT_R8G8B8_SSACLED,</code> <code>VK_FORMAT_R8G8B8_UINT,</code> <code>VK_FORMAT_R8G8B8_SINT,</code> <code>VK_FORMAT_R8G8B8_SRGB,</code> <code>VK_FORMAT_B8G8R8_UNORM,</code> <code>VK_FORMAT_B8G8R8_SNORM,</code> <code>VK_FORMAT_B8G8R8_USCALED,</code> <code>VK_FORMAT_B8G8R8_SSACLED,</code> <code>VK_FORMAT_B8G8R8_UINT,</code> <code>VK_FORMAT_B8G8R8_SINT,</code> <code>VK_FORMAT_B8G8R8_SRGB</code>

Class, Texel Block Size, # Texels/Block	Formats
32-bit Block size 4 bytes 1 texel/block	VK_FORMAT_R8G8B8A8_UNORM, VK_FORMAT_R8G8B8A8_SNORM, VK_FORMAT_R8G8B8A8_USCALED, VK_FORMAT_R8G8B8A8_SSCALED, VK_FORMAT_R8G8B8A8_UINT, VK_FORMAT_R8G8B8A8_SINT, VK_FORMAT_R8G8B8A8_SRGB, VK_FORMAT_B8G8R8A8_UNORM, VK_FORMAT_B8G8R8A8_SNORM, VK_FORMAT_B8G8R8A8_USCALED, VK_FORMAT_B8G8R8A8_SSCALED, VK_FORMAT_B8G8R8A8_UINT, VK_FORMAT_B8G8R8A8_SINT, VK_FORMAT_B8G8R8A8_SRGB, VK_FORMAT_A8B8G8R8_UNORM_PACK32, VK_FORMAT_A8B8G8R8_SNORM_PACK32, VK_FORMAT_A8B8G8R8_USCALED_PACK32, VK_FORMAT_A8B8G8R8_SSCALED_PACK32, VK_FORMAT_A8B8G8R8_UINT_PACK32, VK_FORMAT_A8B8G8R8_SINT_PACK32, VK_FORMAT_A8B8G8R8_SRGB_PACK32, VK_FORMAT_A2R10G10B10_UNORM_PACK32, VK_FORMAT_A2R10G10B10_SNORM_PACK32, VK_FORMAT_A2R10G10B10_USCALED_PACK32, VK_FORMAT_A2R10G10B10_SSCALED_PACK32, VK_FORMAT_A2R10G10B10_UINT_PACK32, VK_FORMAT_A2R10G10B10_SINT_PACK32, VK_FORMAT_A2B10G10R10_UNORM_PACK32, VK_FORMAT_A2B10G10R10_SNORM_PACK32, VK_FORMAT_A2B10G10R10_USCALED_PACK32, VK_FORMAT_A2B10G10R10_SSCALED_PACK32, VK_FORMAT_A2B10G10R10_UINT_PACK32, VK_FORMAT_A2B10G10R10_SINT_PACK32, VK_FORMAT_R16G16_UNORM, VK_FORMAT_R16G16_SNORM, VK_FORMAT_R16G16_USCALED, VK_FORMAT_R16G16_SSCALED, VK_FORMAT_R16G16_UINT, VK_FORMAT_R16G16_SINT, VK_FORMAT_R16G16_SFLOAT, VK_FORMAT_R32_UINT, VK_FORMAT_R32_SINT, VK_FORMAT_R32_SFLOAT, VK_FORMAT_B10G11R11_UFLOAT_PACK32, VK_FORMAT_E5B9G9R9_UFLOAT_PACK32, VK_FORMAT_R10X6G10X6_UNORM_2PACK16, VK_FORMAT_R12X4G12X4_UNORM_2PACK16

Class, Texel Block Size, # Texels/Block	Formats
32-bit G8B8G8R8 Block size 4 bytes 1 texel/block	<code>VK_FORMAT_G8B8G8R8_422_UNORM</code>
32-bit B8G8R8G8 Block size 4 bytes 1 texel/block	<code>VK_FORMAT_B8G8R8G8_422_UNORM</code>
48-bit Block size 6 bytes 1 texel/block	<code>VK_FORMAT_R16G16B16_UNORM,</code> <code>VK_FORMAT_R16G16B16_SNORM,</code> <code>VK_FORMAT_R16G16B16_USCALED,</code> <code>VK_FORMAT_R16G16B16_SSACLED,</code> <code>VK_FORMAT_R16G16B16_UINT,</code> <code>VK_FORMAT_R16G16B16_SINT,</code> <code>VK_FORMAT_R16G16B16_SFLOAT</code>
64-bit Block size 8 bytes 1 texel/block	<code>VK_FORMAT_R16G16B16A16_UNORM,</code> <code>VK_FORMAT_R16G16B16A16_SNORM,</code> <code>VK_FORMAT_R16G16B16A16_USCALED,</code> <code>VK_FORMAT_R16G16B16A16_SSACLED,</code> <code>VK_FORMAT_R16G16B16A16_UINT,</code> <code>VK_FORMAT_R16G16B16A16_SINT,</code> <code>VK_FORMAT_R16G16B16A16_SFLOAT,</code> <code>VK_FORMAT_R32G32_UINT,</code> <code>VK_FORMAT_R32G32_SINT,</code> <code>VK_FORMAT_R32G32_SFLOAT,</code> <code>VK_FORMAT_R64_UINT,</code> <code>VK_FORMAT_R64_SINT,</code> <code>VK_FORMAT_R64_SFLOAT</code>
64-bit R10G10B10A10 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16</code>
64-bit G10B10G10R10 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16</code>
64-bit B10G10R10G10 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16</code>
64-bit R12G12B12A12 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16</code>
64-bit G12B12G12R12 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16</code>
64-bit B12G12R12G12 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16</code>

<b>Class, Texel Block Size, # Texels/Block</b>	<b>Formats</b>
64-bit G16B16G16R16 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_G16B16G16R16_422_UNORM</code>
64-bit B16G16R16G16 Block size 8 bytes 1 texel/block	<code>VK_FORMAT_B16G16R16G16_422_UNORM</code>
96-bit Block size 12 bytes 1 texel/block	<code>VK_FORMAT_R32G32B32_UINT,</code> <code>VK_FORMAT_R32G32B32_SINT,</code> <code>VK_FORMAT_R32G32B32_SFLOAT</code>
128-bit Block size 16 bytes 1 texel/block	<code>VK_FORMAT_R32G32B32A32_UINT,</code> <code>VK_FORMAT_R32G32B32A32_SINT,</code> <code>VK_FORMAT_R32G32B32A32_SFLOAT,</code> <code>VK_FORMAT_R64G64_UINT,</code> <code>VK_FORMAT_R64G64_SINT,</code> <code>VK_FORMAT_R64G64_SFLOAT</code>
192-bit Block size 24 bytes 1 texel/block	<code>VK_FORMAT_R64G64B64_UINT,</code> <code>VK_FORMAT_R64G64B64_SINT,</code> <code>VK_FORMAT_R64G64B64_SFLOAT</code>
256-bit Block size 32 bytes 1 texel/block	<code>VK_FORMAT_R64G64B64A64_UINT,</code> <code>VK_FORMAT_R64G64B64A64_SINT,</code> <code>VK_FORMAT_R64G64B64A64_SFLOAT</code>
BC1_RGB (64 bit) Block size 8 bytes 16 texels/block	<code>VK_FORMAT_BC1_RGB_UNORM_BLOCK,</code> <code>VK_FORMAT_BC1_RGB_SRGB_BLOCK</code>
BC1_RGBA (64 bit) Block size 8 bytes 16 texels/block	<code>VK_FORMAT_BC1_RGBA_UNORM_BLOCK,</code> <code>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</code>
BC2 (128 bit) Block size 16 bytes 16 texels/block	<code>VK_FORMAT_BC2_UNORM_BLOCK,</code> <code>VK_FORMAT_BC2_SRGB_BLOCK</code>
BC3 (128 bit) Block size 16 bytes 16 texels/block	<code>VK_FORMAT_BC3_UNORM_BLOCK,</code> <code>VK_FORMAT_BC3_SRGB_BLOCK</code>
BC4 (64 bit) Block size 8 bytes 16 texels/block	<code>VK_FORMAT_BC4_UNORM_BLOCK,</code> <code>VK_FORMAT_BC4_SNORM_BLOCK</code>
BC5 (128 bit) Block size 16 bytes 16 texels/block	<code>VK_FORMAT_BC5_UNORM_BLOCK,</code> <code>VK_FORMAT_BC5_SNORM_BLOCK</code>
BC6H (128 bit) Block size 16 bytes 16 texels/block	<code>VK_FORMAT_BC6H_UFLOAT_BLOCK,</code> <code>VK_FORMAT_BC6H_SFLOAT_BLOCK</code>

Class, Texel Block Size, # Texels/Block	Formats
BC7 (128 bit) Block size 16 bytes 16 texels/block	VK_FORMAT_BC7_UNORM_BLOCK, VK_FORMAT_BC7_SRGB_BLOCK
ETC2_RGB (64 bit) Block size 8 bytes 16 texels/block	VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK
ETC2_RGBA (64 bit) Block size 8 bytes 16 texels/block	VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK
ETC2_EAC_RGBA (64 bit) Block size 8 bytes 16 texels/block	VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK
EAC_R (64 bit) Block size 8 bytes 16 texels/block	VK_FORMAT_EAC_R11_UNORM_BLOCK, VK_FORMAT_EAC_R11_SNORM_BLOCK
EAC_RG (128 bit) Block size 16 bytes 16 texels/block	VK_FORMAT_EAC_R11G11_UNORM_BLOCK, VK_FORMAT_EAC_R11G11_SNORM_BLOCK
ASTC_4x4 (128 bit) Block size 16 bytes 16 texels/block	VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_4x4_SRGB_BLOCK
ASTC_5x4 (128 bit) Block size 16 bytes 20 texels/block	VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_5x4_SRGB_BLOCK
ASTC_5x5 (128 bit) Block size 16 bytes 25 texels/block	VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_5x5_SRGB_BLOCK
ASTC_6x5 (128 bit) Block size 16 bytes 30 texels/block	VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT,  VK_FORMAT_ASTC_6x5_SRGB_BLOCK
ASTC_6x6 (128 bit) Block size 16 bytes 36 texels/block	VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_6x6_SRGB_BLOCK
ASTC_8x5 (128 bit) Block size 16 bytes 40 texels/block	VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_8x5_SRGB_BLOCK
ASTC_8x6 (128 bit) Block size 16 bytes 48 texels/block	VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_8x6_SRGB_BLOCK
ASTC_8x8 (128 bit) Block size 16 bytes 64 texels/block	VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_8x8_SRGB_BLOCK

<b>Class, Texel Block Size, # Texels/Block</b>	<b>Formats</b>
ASTC_10x5 (128 bit) Block size 16 bytes 50 texels/block	<code>VK_FORMAT_ASTC_10x5_UNORM_BLOCK,</code> <code>VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT,</code> <code>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</code>
ASTC_10x6 (128 bit) Block size 16 bytes 60 texels/block	<code>VK_FORMAT_ASTC_10x6_UNORM_BLOCK,</code> <code>VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT,</code> <code>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</code>
ASTC_10x8 (128 bit) Block size 16 bytes 80 texels/block	<code>VK_FORMAT_ASTC_10x8_UNORM_BLOCK,</code> <code>VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT,</code> <code>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</code>
ASTC_10x10 (128 bit) Block size 16 bytes 100 texels/block	<code>VK_FORMAT_ASTC_10x10_UNORM_BLOCK,</code> <code>VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT,</code> <code>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</code>
ASTC_12x10 (128 bit) Block size 16 bytes 120 texels/block	<code>VK_FORMAT_ASTC_12x10_UNORM_BLOCK,</code> <code>VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT,</code> <code>VK_FORMAT_ASTC_12x10_SRGB_BLOCK</code>
ASTC_12x12 (128 bit) Block size 16 bytes 144 texels/block	<code>VK_FORMAT_ASTC_12x12_UNORM_BLOCK,</code> <code>VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT,</code> <code>VK_FORMAT_ASTC_12x12_SRGB_BLOCK</code>
D16 (16 bit) Block size 2 bytes 1 texel/block	<code>VK_FORMAT_D16_UNORM</code>
D24 (32 bit) Block size 4 bytes 1 texel/block	<code>VK_FORMAT_X8_D24_UNORM_PACK32</code>
D32 (32 bit) Block size 4 bytes 1 texel/block	<code>VK_FORMAT_D32_SFLOAT</code>
S8 (8 bit) Block size 1 byte 1 texel/block	<code>VK_FORMAT_S8_UINT</code>
D16S8 (24 bit) Block size 3 bytes 1 texel/block	<code>VK_FORMAT_D16_UNORM_S8_UINT</code>
D24S8 (32 bit) Block size 4 bytes 1 texel/block	<code>VK_FORMAT_D24_UNORM_S8_UINT</code>
D32S8 (40 bit) Block size 5 bytes 1 texel/block	<code>VK_FORMAT_D32_SFLOAT_S8_UINT</code>
8-bit 3-plane 420 Block size (1,1,1) bytes 1 texel/block	<code>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</code>

Class, Texel Block Size, # Texels/Block	Formats
8-bit 2-plane 420 Block size (1,2) bytes 1 texel/block	<code>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</code>
8-bit 3-plane 422 Block size (1,1,1) bytes 1 texel/block	<code>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</code>
8-bit 2-plane 422 Block size (1,2) bytes 1 texel/block	<code>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</code>
8-bit 3-plane 444 Block size (1,1,1) bytes 1 texel/block	<code>VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM</code>
10-bit 3-plane 420 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</code>
10-bit 2-plane 420 Block size (2,4) bytes 1 texel/block	<code>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</code>
10-bit 3-plane 422 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</code>
10-bit 2-plane 422 Block size (2,2) bytes 1 texel/block	<code>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</code>
10-bit 3-plane 444 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</code>
12-bit 3-plane 420 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</code>
12-bit 2-plane 420 Block size (2,4) bytes 1 texel/block	<code>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</code>
12-bit 3-plane 422 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</code>
12-bit 2-plane 422 Block size (2,4) bytes 1 texel/block	<code>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</code>
12-bit 3-plane 444 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</code>

Class, Texel Block Size, # Texels/Block	Formats
16-bit 3-plane 420 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</code>
16-bit 2-plane 420 Block size (2,4) bytes 1 texel/block	<code>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</code>
16-bit 3-plane 422 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</code>
16-bit 2-plane 422 Block size (2,4) bytes 1 texel/block	<code>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</code>
16-bit 3-plane 444 Block size (2,2,2) bytes 1 texel/block	<code>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</code>

## 37.2. Format Properties

To query supported format features which are properties of the physical device, call:

```
void vkGetPhysicalDeviceFormatProperties(
    VkPhysicalDevice           physicalDevice,
    VkFormat                   format,
    VkFormatProperties*        pFormatProperties);
```

- `physicalDevice` is the physical device from which to query the format properties.
- `format` is the format whose properties are queried.
- `pFormatProperties` is a pointer to a `VkFormatProperties` structure in which physical device properties for `format` are returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `format` **must** be a valid `VkFormat` value
- `pFormatProperties` **must** be a valid pointer to a `VkFormatProperties` structure

The `VkFormatProperties` structure is defined as:

```

typedef struct VkFormatProperties {
    VkFormatFeatureFlags linearTilingFeatures;
    VkFormatFeatureFlags optimalTilingFeatures;
    VkFormatFeatureFlags bufferFeatures;
} VkFormatProperties;

```

- **linearTilingFeatures** is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by images created with a `tiling` parameter of `VK_IMAGE_TILING_LINEAR`.
- **optimalTilingFeatures** is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by images created with a `tiling` parameter of `VK_IMAGE_TILING_OPTIMAL`.
- **bufferFeatures** is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by buffers.

*Note*



If no format feature flags are supported, the format itself is not supported, and images of that format cannot be created.

If `format` is a block-compressed format, then `bufferFeatures` **must** not support any features for the format.

If `format` is not a multi-plane format then `linearTilingFeatures` and `optimalTilingFeatures` **must** not contain `VK_FORMAT_FEATURE_DISJOINT_BIT`.

Bits which **can** be set in the `VkFormatProperties` features `linearTilingFeatures`, `optimalTilingFeatures`, `drmFormatModifierTilingFeatures`, and `bufferFeatures` are:

```

typedef enum VkFormatFeatureFlagBits {
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT = 0x00000001,
    VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT = 0x00000002,
    VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT = 0x00000004,
    VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000008,
    VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT = 0x00000010,
    VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT = 0x00000020,
    VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT = 0x00000040,
    VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT = 0x00000080,
    VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT = 0x00000100,
    VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000200,
    VK_FORMAT_FEATURE_BLIT_SRC_BIT = 0x00000400,
    VK_FORMAT_FEATURE_BLIT_DST_BIT = 0x00000800,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT = 0x00001000,
    VK_FORMAT_FEATURE_TRANSFER_SRC_BIT = 0x00004000,
    VK_FORMAT_FEATURE_TRANSFER_DST_BIT = 0x00008000,
    VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT = 0x00020000,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT = 0x00040000,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT =
    0x00080000,
}

```

```

VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT =
0x00100000,

VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEA
BLE_BIT = 0x00200000,
    VK_FORMAT_FEATURE_DISJOINT_BIT = 0x00400000,
    VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT = 0x00800000,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG = 0x00002000,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT_EXT = 0x00010000,
    VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT = 0x01000000,
    VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR = VK_FORMAT_FEATURE_TRANSFER_SRC_BIT,
    VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR = VK_FORMAT_FEATURE_TRANSFER_DST_BIT,
    VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT_KHR =
VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT_KHR =
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT,

VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT_KH
R =
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT,

VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT_KH
R =
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT,

VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEA
BLE_BIT_KHR =
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEA
BLE_BIT,
    VK_FORMAT_FEATURE_DISJOINT_BIT_KHR = VK_FORMAT_FEATURE_DISJOINT_BIT,
    VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT_KHR =
VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT =
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG,
    VK_FORMAT_FEATURE_FLAG_BITS_MAX_ENUM = 0xFFFFFFFF
} VkFormatFeatureFlagBits;

```

The following bits **may** be set in `linearTilingFeatures`, `optimalTilingFeatures`, and `drmFormatModifierTilingFeatures`, specifying that the features are supported by `images` or `image views` created with the queried `vkGetPhysicalDeviceFormatProperties::format`:

- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` specifies that an image view **can** be `sampled` from.
- `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT` specifies that an image view **can** be used as a `storage images`.
- `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT` specifies that an image view **can** be used as storage image that supports atomic operations.
- `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` specifies that an image view **can** be used as a framebuffer color attachment and as an input attachment.

- `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT` specifies that an image view **can** be used as a framebuffer color attachment that supports blending and as an input attachment.
- `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT` specifies that an image view **can** be used as a framebuffer depth/stencil attachment and as an input attachment.
- `VK_FORMAT_FEATURE_BLIT_SRC_BIT` specifies that an image **can** be used as `srcImage` for the `vkCmdBlitImage` command.
- `VK_FORMAT_FEATURE_BLIT_DST_BIT` specifies that an image **can** be used as `dstImage` for the `vkCmdBlitImage` command.
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` specifies that if `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` is also set, an image view **can** be used with a sampler that has either of `magFilter` or `minFilter` set to `VK_FILTER_LINEAR`, or `mipMapMode` set to `VK_SAMPLER_MIPMAP_MODE_LINEAR`. If `VK_FORMAT_FEATURE_BLIT_SRC_BIT` is also set, an image can be used as the `srcImage` to `vkCmdBlitImage` with a `filter` of `VK_FILTER_LINEAR`. This bit **must** only be exposed for formats that also support the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` or `VK_FORMAT_FEATURE_BLIT_SRC_BIT`.

If the format being queried is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports linear filtering, and that linear filtering of the depth aspect is supported whether depth compare is enabled in the sampler or not. If this bit is not present, linear filtering with depth compare disabled is unsupported and linear filtering with depth compare enabled is supported, but **may** compute the filtered value in an implementation-dependent manner which differs from the normal rules of linear filtering. The resulting value **must** be in the range [0,1] and **should** be proportional to, or a weighted average of, the number of comparison passes or failures.

- `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT` specifies that an image **can** be used as a source image for `copy` commands.
- `VK_FORMAT_FEATURE_TRANSFER_DST_BIT` specifies that an image **can** be used as a destination image for `copy` commands and `clear` commands.
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT_EXT` specifies `VkImage` **can** be used as a sampled image with a min or max `VkSamplerReductionModeEXT`. This bit **must** only be exposed for formats that also support the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`.
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT` specifies that `VkImage` **can** be used with a sampler that has either of `magFilter` or `minFilter` set to `VK_FILTER_CUBIC_EXT`, or be the source image for a blit with `filter` set to `VK_FILTER_CUBIC_EXT`. This bit **must** only be exposed for formats that also support the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`. If the format being queried is a depth/stencil format, this only specifies that the depth aspect is cubic filterable.
- `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT` specifies that an application **can** define a sampler Y'CbCr conversion using this format as a source, and that an image of this format **can** be used with a `VkSamplerYcbcrConversionCreateInfo` `xChromaOffset` and/or `yChromaOffset` of `VK_CHROMA_LOCATION_MIDPOINT`. Otherwise both `xChromaOffset` and `yChromaOffset` **must** be `VK_CHROMA_LOCATION_COSITED_EVEN`. If a format does not incorporate chroma downsampling (it is not a “422” or “420” format) but the implementation supports sampler Y'CbCr conversion for this format, the implementation **must** set `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT`.
- `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT` specifies that an application **can** define a

sampler Y'CbCr conversion using this format as a source, and that an image of this format **can** be used with a `VkSamplerYcbcrConversionCreateInfo` `xChromaOffset` and/or `yChromaOffset` of `VK_CHROMA_LOCATION_COSITED_EVEN`. Otherwise both `xChromaOffset` and `yChromaOffset` **must** be `VK_CHROMA_LOCATION_MIDPOINT`. If neither `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT` nor `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT` is set, the application **must** not define a sampler Y'CbCr conversion using this format as a source.

- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT` specifies that the format can do linear sampler filtering (min/magFilter) whilst sampler Y'CbCr conversion is enabled.
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT` specifies that the format can have different chroma, min, and mag filters.
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT` specifies that reconstruction is explicit, as described in [Chroma Reconstruction](#). If this bit is not present, reconstruction is implicit by default.
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT` specifies that reconstruction **can** be forcibly made explicit by setting `VkSamplerYcbcrConversionCreateInfo::forceExplicitReconstruction` to `VK_TRUE`.
- `VK_FORMAT_FEATURE_DISJOINT_BIT` specifies that a multi-planar image **can** have the `VK_IMAGE_CREATE_DISJOINT_BIT` set during image creation. An implementation **must** not set `VK_FORMAT_FEATURE_DISJOINT_BIT` for *single-plane formats*.
- `VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT` specifies that an image view **can** be used as a fragment density map attachment.

The following bits **may** be set in `bufferFeatures`, specifying that the features are supported by buffers or buffer views created with the queried `vkGetPhysicalDeviceProperties::format`:

- `VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT` specifies that the format **can** be used to create a buffer view that **can** be bound to a `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` descriptor.
- `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` specifies that the format **can** be used to create a buffer view that **can** be bound to a `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` descriptor.
- `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT` specifies that atomic operations are supported on `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` with this format.
- `VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT` specifies that the format **can** be used as a vertex attribute format (`VkVertexInputAttributeDescription::format`).

```
typedef VkFlags VkFormatFeatureFlags;
```

`VkFormatFeatureFlags` is a bitmask type for setting a mask of zero or more `VkFormatFeatureFlagBits`.

To query supported format features which are properties of the physical device, call:

```
void vkGetPhysicalDeviceFormatProperties2(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkFormatProperties2* pFormatProperties);
```

or the equivalent command

```
void vkGetPhysicalDeviceFormatProperties2KHR(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkFormatProperties2* pFormatProperties);
```

- `physicalDevice` is the physical device from which to query the format properties.
- `format` is the format whose properties are queried.
- `pFormatProperties` is a pointer to a `VkFormatProperties2` structure in which physical device properties for `format` are returned.

`vkGetPhysicalDeviceFormatProperties2` behaves similarly to `vkGetPhysicalDeviceFormatProperties`, with the ability to return extended information in a `pNext` chain of output structures.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `format` **must** be a valid `VkFormat` value
- `pFormatProperties` **must** be a valid pointer to a `VkFormatProperties2` structure

The `VkFormatProperties2` structure is defined as:

```
typedef struct VkFormatProperties2 {
    VkStructureType sType;
    void* pNext;
    VkFormatProperties formatProperties;
} VkFormatProperties2;
```

or the equivalent

```
typedef VkFormatProperties2 VkFormatProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `formatProperties` is a `VkFormatProperties` structure describing features supported by the requested format.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2`
- `pNext` must be `NULL` or a pointer to a valid instance of `VkDrmFormatModifierPropertiesListEXT`

To obtain the list of `Linux DRM format modifiers` compatible with a `VkFormat`, add `VkDrmFormatModifierPropertiesListEXT` to the `pNext` chain of `VkFormatProperties2`.

The `VkDrmFormatModifierPropertiesListEXT` structure is defined as:

```
typedef struct VkDrmFormatModifierPropertiesListEXT {  
    VkStructureType sType;  
    void* pNext;  
    uint32_t drmFormatModifierCount;  
    VkDrmFormatModifierPropertiesEXT* pDrmFormatModifierProperties;  
} VkDrmFormatModifierPropertiesListEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `drmFormatModifierCount` is an inout parameter related to the number of modifiers compatible with the `format`, as described below.
- `pDrmFormatModifierProperties` is either `NULL` or an array of `VkDrmFormatModifierPropertiesEXT` structures.

If `pDrmFormatModifierProperties` is `NULL`, then the function returns in `drmFormatModifierCount` the number of modifiers compatible with the queried `format`. Otherwise, the application must set `drmFormatModifierCount` to the length of the array `pDrmFormatModifierProperties`; the function will write at most `drmFormatModifierCount` elements to the array, and will return in `drmFormatModifierCount` the number of elements written.

Among the elements in array `pDrmFormatModifierProperties`, each returned `drmFormatModifier` must be unique.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT`

The `VkDrmFormatModifierPropertiesEXT` structure describes properties of a `VkFormat` when that format is combined with a `Linux DRM format modifier`. These properties, like those of `VkFormatProperties2`, are independent of any particular image.

The `VkDrmFormatModifierPropertiesEXT` structure is defined as:

```

typedef struct VkDrmFormatModifierPropertiesEXT {
    uint64_t          drmFormatModifier;
    uint32_t          drmFormatModifierPlaneCount;
    VkFormatFeatureFlags  drmFormatModifierTilingFeatures;
} VkDrmFormatModifierPropertiesEXT;

```

- `drmFormatModifier` is a *Linux DRM format modifier*.
- `drmFormatModifierPlaneCount` is the number of *memory planes* in any image created with `format` and `drmFormatModifier`. An image's *memory planecount* is distinct from its *format planecount*, as explained below.
- `drmFormatModifierTilingFeatures` is a bitmask of `VkFormatFeatureFlagBits` that are supported by any image created with `format` and `drmFormatModifier`.

The returned `drmFormatModifierTilingFeatures` **must** contain at least one bit.

The implementation **must** not return `DRM_FORMAT_MOD_INVALID` in `drmFormatModifier`.

An image's *memory planecount* (as returned by `drmFormatModifierPlaneCount`) is distinct from its *format planecount* (in the sense of multi-planar  $\text{Y'CB}_\text{R}$  formats). In `VkImageAspectFlags`, each `VK_IMAGE_ASPECT_MEMORY_PLANEi_BIT_EXT` represents a *memory plane* and each `VK_IMAGE_ASPECT_PLANEi_BIT` a *format plane*.

An image's set of *format planes* is an ordered partition of the image's **content** into separable groups of format channels. The ordered partition is encoded in the name of each `VkFormat`. For example, `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM` contains two *format planes*; the first plane contains the green channel and the second plane contains the blue channel and red channel. If the format name does not contain `PLANE`, then the format contains a single plane; for example, `VK_FORMAT_R8G8B8A8_UNORM`. Some commands, such as `vkCmdCopyBufferToImage`, do not operate on all format channels in the image, but instead operate only on the *format planes* explicitly chosen by the application and operate on each *format plane* independently.

An image's set of *memory planes* is an ordered partition of the image's **memory** rather than the image's **content**. Each *memory plane* is a contiguous range of memory. The union of an image's *memory planes* is not necessarily contiguous.

If an image is `linear`, then the partition is the same for *memory planes* and for *format planes*. Therefore, if the returned `drmFormatModifier` is `DRM_FORMAT_MOD_LINEAR`, then `drmFormatModifierPlaneCount` **must** equal the *format planecount*, and `drmFormatModifierTilingFeatures` **must** be identical to the `VkFormatProperties2::linearTilingFeatures` returned in the same `pNext` chain.

If an image is `non-linear`, then the partition of the image's **memory** into *memory planes* is implementation-specific and **may** be unrelated to the partition of the image's **content** into *format planes*. For example, consider an image whose `format` is `VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM`, `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, whose `drmFormatModifier` is not `DRM_FORMAT_MOD_LINEAR`, and `flags` lacks `VK_IMAGE_CREATE_DISJOINT_BIT`. The image has 3 *format planes*, and commands such as `vkCmdCopyBufferToImage` act on each *format plane* independently as if the data of each *format plane* were separable from the data of the other planes. In a

straightforward implementation, the implementation **may** store the image's content in 3 adjacent *memory planes* where each *memory plane* corresponds exactly to a *format plane*. However, the implementation **may** also store the image's content in a single *memory plane* where all format channels are combined using an implementation-private block-compressed format; or the implementation **may** store the image's content in a collection of 7 adjacent *memory planes* using an implementation-private sharding technique. Because the image is non-linear and non-disjoint, the implementation has much freedom when choosing the image's placement in memory.

The *memory planecount* applies to function parameters and structures only when the API specifies an explicit requirement on `dmFormatModifierPlaneCount`. In all other cases, the *memory planecount* is ignored.

### 37.3. Required Format Support

Implementations **must** support at least the following set of features on the listed formats. For images, these features **must** be supported for every `VkImageType` (including arrayed and cube variants) unless otherwise noted. These features are supported on existing formats without needing to advertise an extension or needing to explicitly enable them. Support for additional functionality beyond the requirements listed here is queried using the `vkGetPhysicalDeviceFormatProperties` command.



*Note*

Unless otherwise excluded below, the required formats are supported for all `VkImageCreateFlags` values as long as those flag values are otherwise allowed.

The following tables show which feature bits **must** be supported for each format. Formats that are required to support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` **must** also support `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT` and `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.

Table 55. Key for format feature tables

✓	This feature <b>must</b> be supported on the named format
†	This feature <b>must</b> be supported on at least some of the named formats, with more information in the table where the symbol appears

Table 56. Feature bits in `optimalTilingFeatures`

<code>VK_FORMAT_FEATURE_TRANSFER_SRC_BIT</code>
<code>VK_FORMAT_FEATURE_TRANSFER_DST_BIT</code>
<code>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</code>
<code>VK_FORMAT_FEATURE_BLIT_SRC_BIT</code>
<code>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</code>
<code>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</code>
<code>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</code>
<code>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</code>
<code>VK_FORMAT_FEATURE_BLIT_DST_BIT</code>
<code>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</code>

VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT_EXT

Table 57. Feature bits in `bufferFeatures`

VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT
VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT
VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT
VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT

Table 58. Mandatory format support: sub-byte channels

Format	VK_FORMAT_UNDEFINED	VK_FORMAT_R4G4_UNORM_PACK8	VK_FORMAT_R4G4B4A4_UNORM_PACK16	VK_FORMAT_B4G4R4A4_UNORM_PACK16	VK_FORMAT_R5G6B5_UNORM_PACK16	VK_FORMAT_B5G6R5_UNORM_PACK16	VK_FORMAT_R5G5B5A1_UNORM_PACK16	VK_FORMAT_B5G5R5A1_UNORM_PACK16	VK_FORMAT_A1R5G5B5_UNORM_PACK16	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT	VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT	VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT	VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT	VK_FORMAT_FEATURE_BLIT_DST_BIT	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT	VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT	VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT	VK_FORMAT_FEATURE_BLIT_SRC_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT
VK_FORMAT_UNDEFINED										↙												
VK_FORMAT_R4G4_UNORM_PACK8											↙											
VK_FORMAT_R4G4B4A4_UNORM_PACK16												↙										
VK_FORMAT_B4G4R4A4_UNORM_PACK16				✓	✓	✓																
VK_FORMAT_R5G6B5_UNORM_PACK16				✓	✓	✓						✓	✓	✓								
VK_FORMAT_B5G6R5_UNORM_PACK16																						
VK_FORMAT_R5G5B5A1_UNORM_PACK16																						
VK_FORMAT_B5G5R5A1_UNORM_PACK16																						
VK_FORMAT_A1R5G5B5_UNORM_PACK16				✓	✓	✓						✓	✓	✓								

Table 59. Mandatory format support: 1-3 byte-sized channels

	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT							
	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT							
	VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT							
	VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT							
	VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT							
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT							
	VK_FORMAT_FEATURE_BLIT_DST_BIT							
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT							
	VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT							
	VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT							
	VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT							
	VK_FORMAT_FEATURE_BLIT_SRC_BIT							
	VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT							
<b>Format</b>								
VK_FORMAT_R8_UNORM	✓	✓	✓			✓	✓	✓
VK_FORMAT_R8_SNORM	✓	✓	✓					✓
VK_FORMAT_R8_USCALED								
VK_FORMAT_R8_SSACLED								
VK_FORMAT_R8_UINT	✓	✓				✓	✓	✓
VK_FORMAT_R8_SINT	✓	✓				✓	✓	✓
VK_FORMAT_R8_SRGB								
VK_FORMAT_R8G8_UNORM	✓	✓	✓			✓	✓	✓
VK_FORMAT_R8G8_SNORM	✓	✓	✓					✓
VK_FORMAT_R8G8_USCALED								
VK_FORMAT_R8G8_SSACLED								
VK_FORMAT_R8G8_UINT	✓	✓				✓	✓	✓
VK_FORMAT_R8G8_SINT	✓	✓				✓	✓	✓
VK_FORMAT_R8G8_SRGB								
VK_FORMAT_R8G8B8_UNORM								
VK_FORMAT_R8G8B8_SNORM								
VK_FORMAT_R8G8B8_USCALED								
VK_FORMAT_R8G8B8_SSACLED								
VK_FORMAT_R8G8B8_UINT								
VK_FORMAT_R8G8B8_SINT								
VK_FORMAT_R8G8B8_SRGB								
VK_FORMAT_B8G8R8_UNORM								
VK_FORMAT_B8G8R8_SNORM								
VK_FORMAT_B8G8R8_USCALED								
VK_FORMAT_B8G8R8_SSACLED								
VK_FORMAT_B8G8R8_UINT								
VK_FORMAT_B8G8R8_SINT								
VK_FORMAT_B8G8R8_SRGB								

Table 60. Mandatory format support: 4 byte-sized channels

Format	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT										
	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT										
VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT											
VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT											
VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT											
VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT											
VK_FORMAT_FEATURE_BLIT_DST_BIT											
VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT											
VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT											
VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT											
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT											
VK_FORMAT_FEATURE_BLIT_SRC_BIT											
VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT											
VK_FORMAT_R8G8B8A8_UNORM	✓	✓	✓	✓			✓	✓	✓	✓	✓
VK_FORMAT_R8G8B8A8_SNORM	✓	✓	✓	✓						✓	✓
VK_FORMAT_R8G8B8A8_USCALED											
VK_FORMAT_R8G8B8A8_SSCALED											
VK_FORMAT_R8G8B8A8_UINT	✓	✓			✓		✓	✓		✓	✓
VK_FORMAT_R8G8B8A8_SINT	✓	✓			✓		✓	✓		✓	✓
VK_FORMAT_R8G8B8A8_SRGB	✓	✓	✓				✓	✓	✓		
VK_FORMAT_B8G8R8A8_UNORM	✓	✓	✓				✓	✓	✓	✓	✓
VK_FORMAT_B8G8R8A8_SNORM											
VK_FORMAT_B8G8R8A8_USCALED											
VK_FORMAT_B8G8R8A8_SSCALED											
VK_FORMAT_B8G8R8A8_UINT											
VK_FORMAT_B8G8R8A8_SINT											
VK_FORMAT_B8G8R8A8_SRGB	✓	✓	✓				✓	✓	✓		
VK_FORMAT_A8B8G8R8_UNORM_PACK32	✓	✓	✓				✓	✓	✓	✓	✓
VK_FORMAT_A8B8G8R8_SNORM_PACK32	✓	✓	✓							✓	✓
VK_FORMAT_A8B8G8R8_USCALED_PACK32											
VK_FORMAT_A8B8G8R8_SSCALED_PACK32											
VK_FORMAT_A8B8G8R8_UINT_PACK32	✓	✓					✓	✓		✓	✓
VK_FORMAT_A8B8G8R8_SINT_PACK32	✓	✓					✓	✓		✓	✓
VK_FORMAT_A8B8G8R8_SRGB_PACK32	✓	✓	✓				✓	✓	✓		

Table 61. Mandatory format support: 10- and 12-bit channels

Format	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT	VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT	VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT	VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT	VK_FORMAT_FEATURE_BLIT_DST_BIT	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT	VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT	VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT	VK_FORMAT_FEATURE_BLIT_SRC_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT
VK_FORMAT_A2R10G10B10_UNORM_PACK32													
VK_FORMAT_A2R10G10B10_SNORM_PACK32													
VK_FORMAT_A2R10G10B10_USCALED_PACK32													
VK_FORMAT_A2R10G10B10_SSCALED_PACK32													
VK_FORMAT_A2R10G10B10_UINT_PACK32													
VK_FORMAT_A2R10G10B10_SINT_PACK32													
VK_FORMAT_A2B10G10R10_UNORM_PACK32	✓	✓	✓				✓	✓	✓	✓	✓	✓	✓
VK_FORMAT_A2B10G10R10_SNORM_PACK32													
VK_FORMAT_A2B10G10R10_USCALED_PACK32													
VK_FORMAT_A2B10G10R10_SSCALED_PACK32													
VK_FORMAT_A2B10G10R10_UINT_PACK32	✓	✓					✓	✓					✓
VK_FORMAT_A2B10G10R10_SINT_PACK32													
VK_FORMAT_R10X6_UNORM_PACK16													
VK_FORMAT_R10X6G10X6_UNORM_2PACK16													
VK_FORMAT_R12X4_UNORM_PACK16													
VK_FORMAT_R12X4G12X4_UNORM_2PACK16													

Table 62. Mandatory format support: 16-bit channels

	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT		VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT		VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT		VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT		VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT		VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT		VK_FORMAT_FEATURE_BLIT_DST_BIT		VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT		VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT		VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT		VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT		VK_FORMAT_FEATURE_BLIT_SRC_BIT		VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT		Format
VK_FORMAT_R16_UNORM	✓																										
VK_FORMAT_R16_SNORM		✓																									
VK_FORMAT_R16_USCALED																											
VK_FORMAT_R16_SSACLED																											
VK_FORMAT_R16_UINT	✓	✓																									
VK_FORMAT_R16_SINT	✓	✓																									
VK_FORMAT_R16_SFLOAT	✓	✓	✓																								
VK_FORMAT_R16G16_UNORM																										✓	
VK_FORMAT_R16G16_SNORM																										✓	
VK_FORMAT_R16G16_USCALED																											
VK_FORMAT_R16G16_SSACLED																											
VK_FORMAT_R16G16_UINT	✓	✓																								✓	
VK_FORMAT_R16G16_SINT	✓	✓																								✓	
VK_FORMAT_R16G16_SFLOAT	✓	✓	✓																							✓	
VK_FORMAT_R16G16B16_UNORM																											
VK_FORMAT_R16G16B16_SNORM																											
VK_FORMAT_R16G16B16_USCALED																											
VK_FORMAT_R16G16B16_SSACLED																											
VK_FORMAT_R16G16B16_UINT																											
VK_FORMAT_R16G16B16_SINT																											
VK_FORMAT_R16G16B16_SFLOAT																											
VK_FORMAT_R16G16B16A16_UNORM																										✓	
VK_FORMAT_R16G16B16A16_SNORM																										✓	
VK_FORMAT_R16G16B16A16_USCALED																											
VK_FORMAT_R16G16B16A16_SSACLED																											
VK_FORMAT_R16G16B16A16_UINT	✓	✓																								✓	
VK_FORMAT_R16G16B16A16_SINT	✓	✓																								✓	
VK_FORMAT_R16G16B16A16_SFLOAT	✓	✓	✓	✓																						✓	

Table 63. Mandatory format support: 32-bit channels

Format	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT											
	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT											
	VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT											
	VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT											
	VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT											
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT											
	VK_FORMAT_FEATURE_BLIT_DST_BIT											
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT											
	VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT											
	VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT											
VK_FORMAT_R32_UINT	✓	✓			✓	✓	✓	✓			✓	✓
VK_FORMAT_R32_SINT	✓	✓			✓	✓	✓	✓			✓	✓
VK_FORMAT_R32_SFLOAT	✓	✓			✓		✓	✓			✓	✓
VK_FORMAT_R32G32_UINT	✓	✓			✓		✓	✓			✓	✓
VK_FORMAT_R32G32_SINT	✓	✓			✓		✓	✓			✓	✓
VK_FORMAT_R32G32_SFLOAT	✓	✓			✓		✓	✓			✓	✓
VK_FORMAT_R32G32B32_UINT												✓
VK_FORMAT_R32G32B32_SINT												✓
VK_FORMAT_R32G32B32_SFLOAT												✓
VK_FORMAT_R32G32B32A32_UINT	✓	✓			✓		✓	✓			✓	✓
VK_FORMAT_R32G32B32A32_SINT	✓	✓			✓		✓	✓			✓	✓
VK_FORMAT_R32G32B32A32_SFLOAT	✓	✓			✓		✓	✓			✓	✓

Table 64. Mandatory format support: 64-bit/uneven channels

Format	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT	VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT	VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT	VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT	VK_FORMAT_FEATURE_BLIT_DST_BIT	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT	VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT	VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT	VK_FORMAT_FEATURE_BLIT_SRC_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT
VK_FORMAT_R64_UINT													
VK_FORMAT_R64_SINT													
VK_FORMAT_R64_SFLOAT													
VK_FORMAT_R64G64_UINT													
VK_FORMAT_R64G64_SINT													
VK_FORMAT_R64G64_SFLOAT													
VK_FORMAT_R64G64B64_UINT													
VK_FORMAT_R64G64B64_SINT													
VK_FORMAT_R64G64B64_SFLOAT													
VK_FORMAT_R64G64B64A64_UINT													
VK_FORMAT_R64G64B64A64_SINT													
VK_FORMAT_R64G64B64A64_SFLOAT													
VK_FORMAT_B10G11R11_UFLOAT_PACK32	✓	✓	✓										✓
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32	✓	✓	✓										

Table 65. Mandatory format support: depth/stencil with VkImageType VK\_IMAGE\_TYPE\_2D

Format	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT							
	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT							
VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT								
VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT								
VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT								
VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT								
VK_FORMAT_FEATURE_BLIT_DST_BIT								
VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT								
VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT								
VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT								
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT								
VK_FORMAT_FEATURE_BLIT_SRC_BIT								
VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT								
VK_FORMAT_D16_UNORM	✓	✓						✓
VK_FORMAT_X8_D24_UNORM_PACK32								†
VK_FORMAT_D32_SFLOAT	✓	✓						†
VK_FORMAT_S8_UINT								
VK_FORMAT_D16_UNORM_S8_UINT								
VK_FORMAT_D24_UNORM_S8_UINT								†
VK_FORMAT_D32_SFLOAT_S8_UINT								†
VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT feature <b>must</b> be supported for at least one of VK_FORMAT_X8_D24_UNORM_PACK32 and VK_FORMAT_D32_SFLOAT, and <b>must</b> be supported for at least one of VK_FORMAT_D24_UNORM_S8_UINT and VK_FORMAT_D32_SFLOAT_S8_UINT.								

Table 66. Mandatory format support: BC compressed formats with `VkImageType VK_IMAGE_TYPE_2D` and `VK_IMAGE_TYPE_3D`

Format	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT		
	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT		
	VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT		
	VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT		
	VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT		
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT		
	VK_FORMAT_FEATURE_BLIT_DST_BIT		
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT		
	VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT		
	VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT		
<code>VK_FORMAT_BC1_RGB_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC1_RGB_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC2_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC2_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC3_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC3_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC4_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC4_SNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC5_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC5_SNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC6H_UFLOAT_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC6H_SFLOAT_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC7_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_BC7_SRGB_BLOCK</code>	†	†	†

The `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features **must** be supported in `optimalTilingFeatures` for all the formats in at least one of: this table, [Mandatory format support: ETC2 and EAC compressed formats with `VkImageType VK\_IMAGE\_TYPE\_2D`](#), or [Mandatory format support: ASTC LDR compressed formats with `VkImageType VK\_IMAGE\_TYPE\_2D`](#).

Table 67. Mandatory format support: ETC2 and EAC compressed formats with `VkImageType VK_IMAGE_TYPE_2D`

Format	<code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code>	<code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</code>	<code>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</code>	<code>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</code>	<code>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</code>	<code>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</code>	<code>VK_FORMAT_FEATURE_BLIT_DST_BIT</code>	<code>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</code>	<code>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</code>	<code>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</code>	<code>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</code>	<code>VK_FORMAT_FEATURE_BLIT_SRC_BIT</code>	<code>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</code>
<code>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK</code>	†	†	†										
<code>VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</code>	†	†	†										
<code>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK</code>	†	†	†										
<code>VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</code>	†	†	†										
<code>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK</code>	†	†	†										
<code>VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</code>	†	†	†										
<code>VK_FORMAT_EAC_R11_UNORM_BLOCK</code>	†	†	†										
<code>VK_FORMAT_EAC_R11_SNORM_BLOCK</code>	†	†	†										
<code>VK_FORMAT_EAC_R11G11_UNORM_BLOCK</code>	†	†	†										
<code>VK_FORMAT_EAC_R11G11_SNORM_BLOCK</code>	†	†	†										

The `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features **must** be supported in `optimalTilingFeatures` for all the formats in at least one of: this table, [Mandatory format support: BC compressed formats with `VkImageType VK\_IMAGE\_TYPE\_2D` and `VK\_IMAGE\_TYPE\_3D`](#), or [Mandatory format support: ASTC LDR compressed formats with `VkImageType VK\_IMAGE\_TYPE\_2D`](#).

Table 68. Mandatory format support: ASTC LDR compressed formats with `VkImageType VK_IMAGE_TYPE_2D`

Format	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT		
	VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT		
Format	VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT		
	VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT		
Format	VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT		
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT		
Format	VK_FORMAT_FEATURE_BLIT_DST_BIT		
	VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT		
Format	VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT		
	VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT		
Format	VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT		
	VK_FORMAT_FEATURE_BLIT_SRC_BIT		
Format	VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT		
	↓	↓	↓
<code>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x10_UNORM_BLOCK</code>	†	†	†
<code>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</code>	†	†	†

VK_FORMAT_ASTC_12x10_UNORM_BLOCK	†	†	†								
VK_FORMAT_ASTC_12x10_SRGB_BLOCK	†	†	†								
VK_FORMAT_ASTC_12x12_UNORM_BLOCK	†	†	†								
VK_FORMAT_ASTC_12x12_SRGB_BLOCK	†	†	†								

The `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features **must** be supported in `optimalTilingFeatures` for all the formats in at least one of: this table, **Mandatory format support:** BC compressed formats with `VkImageType VK_IMAGE_TYPE_2D` and `VK_IMAGE_TYPE_3D`, or **Mandatory format support:** ETC2 and EAC compressed formats with `VkImageType VK_IMAGE_TYPE_2D`.

If cubic filtering is supported, `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT` **must** be supported for the following image view types:

- `VK_IMAGE_VIEW_TYPE_2D`
- `VK_IMAGE_VIEW_TYPE_2D_ARRAY`

for the following formats:

- `VK_FORMAT_R4G4_UNORM_PACK8`
- `VK_FORMAT_R4G4B4A4_UNORM_PACK16`
- `VK_FORMAT_B4G4R4A4_UNORM_PACK16`
- `VK_FORMAT_R5G6B5_UNORM_PACK16`
- `VK_FORMAT_B5G6R5_UNORM_PACK16`
- `VK_FORMAT_R5G5B5A1_UNORM_PACK16`
- `VK_FORMAT_B5G5R5A1_UNORM_PACK16`
- `VK_FORMAT_A1R5G5B5_UNORM_PACK16`
- `VK_FORMAT_R8_UNORM`
- `VK_FORMAT_R8_SNORM`
- `VK_FORMAT_R8_SRGB`
- `VK_FORMAT_R8G8_UNORM`
- `VK_FORMAT_R8G8_SNORM`
- `VK_FORMAT_R8G8_SRGB`
- `VK_FORMAT_R8G8B8_UNORM`
- `VK_FORMAT_R8G8B8_SNORM`
- `VK_FORMAT_R8G8B8_SRGB`
- `VK_FORMAT_B8G8R8_UNORM`
- `VK_FORMAT_B8G8R8_SNORM`
- `VK_FORMAT_B8G8R8_SRGB`
- `VK_FORMAT_R8G8B8A8_UNORM`
- `VK_FORMAT_R8G8B8A8_SNORM`
- `VK_FORMAT_R8G8B8A8_SRGB`
- `VK_FORMAT_B8G8R8A8_UNORM`
- `VK_FORMAT_B8G8R8A8_SNORM`
- `VK_FORMAT_B8G8R8A8_SRGB`

- VK\_FORMAT\_A8B8G8R8\_UNORM\_PACK32
- VK\_FORMAT\_A8B8G8R8\_SNORM\_PACK32
- VK\_FORMAT\_A8B8G8R8\_USCALED\_PACK32
- VK\_FORMAT\_A8B8G8R8\_SSACLED\_PACK32
- VK\_FORMAT\_A8B8G8R8\_UINT\_PACK32
- VK\_FORMAT\_A8B8G8R8\_SINT\_PACK32
- VK\_FORMAT\_A8B8G8R8\_SRGB\_PACK32

If ETC compressed formats are supported, `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT` must be supported for the following image view types:

- VK\_IMAGE\_VIEW\_TYPE\_2D
- VK\_IMAGE\_VIEW\_TYPE\_2D\_ARRAY

for the following additional formats:

- VK\_FORMAT\_ETC2\_R8G8B8\_UNORM\_BLOCK
- VK\_FORMAT\_ETC2\_R8G8B8\_SRGB\_BLOCK
- VK\_FORMAT\_ETC2\_R8G8B8A1\_UNORM\_BLOCK
- VK\_FORMAT\_ETC2\_R8G8B8A1\_SRGB\_BLOCK
- VK\_FORMAT\_ETC2\_R8G8B8A8\_UNORM\_BLOCK
- VK\_FORMAT\_ETC2\_R8G8B8A8\_SRGB\_BLOCK

If cubic filtering is supported for any other formats, the following image view types must be supported for those formats:

- VK\_IMAGE\_VIEW\_TYPE\_2D
- VK\_IMAGE\_VIEW\_TYPE\_2D\_ARRAY

To be used with `VkImageView` with `subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT`, `sampler Y'CbCr conversion` must be enabled for the following formats:

*Table 69. Formats requiring sampler Y'CbCr conversion for VK\_IMAGE\_ASPECT\_COLOR\_BIT image views*

Format	Planes	VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT	VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT	VK_FORMAT_FEATURE_COISITED_CHROMA_SAMPLES_BIT	VK_FORMAT_FEATURE_TRANSFER_DST_BIT	VK_FORMAT_FEATURE_TRANSFER_SRC_BIT	VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT	VK_FORMAT_FEATURE_DISJOINT_BIT
VK_FORMAT_G8B8G8R8_422_UNORM	1										
VK_FORMAT_B8G8R8G8_422_UNORM	1										

VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM	3	✓	✓	✓	✓				
VK_FORMAT_G8_B8R8_2PLANE_420_UNORM	2		✓	✓	✓	✓			
VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM	3								
VK_FORMAT_G8_B8R8_2PLANE_422_UNORM	2								
VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM	3								
VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16	1								
VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16	1								
VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16	1								
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16	3								
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16	2								
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16	3								
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16	2								
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16	3								
VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16	1								
VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16	1								
VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16	1								
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16	3								
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16	2								
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16	3								
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16	2								
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16	3								
VK_FORMAT_G16B16G16R16_422_UNORM	1								
VK_FORMAT_B16G16R16G16_422_UNORM	1								
VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM	3								
VK_FORMAT_G16_B16R16_2PLANE_420_UNORM	2								
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM	3								
VK_FORMAT_G16_B16R16_2PLANE_422_UNORM	2								
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM	3								

Format features marked **✓ must** be supported only if

[VkPhysicalDeviceSamplerYcbcrConversionFeatures](#) is enabled, and only with [VkImageType\\_VK\\_IMAGE\\_TYPE\\_2D](#)

Implementations are not required to support the [VK\\_IMAGE\\_CREATE\\_SPARSE\\_BINDING\\_BIT](#), [VK\\_IMAGE\\_CREATE\\_SPARSE\\_RESIDENCY\\_BIT](#), or [VK\\_IMAGE\\_CREATE\\_SPARSE\\_ALIASED\\_BIT](#) [VkImageCreateFlags](#) for the above formats that require sampler Y'CbCr conversion. To determine whether the implementation supports sparse image creation flags with these formats use [vkGetPhysicalDeviceImageFormatProperties](#) or [vkGetPhysicalDeviceImageFormatProperties2](#).

[VK\\_FORMAT\\_FEATURE\\_FRAGMENT\\_DENSITY\\_MAP\\_BIT\\_EXT](#) must be supported for the following formats if the fragment density map feature is enabled:

- `VK_FORMAT_R8G8_UNORM`

# Chapter 38. Additional Capabilities

This chapter describes additional capabilities beyond the minimum capabilities described in the [\(Limits and Formats](#) chapters, including:

- [Additional Image Capabilities](#)
- [Additional Buffer Capabilities](#)
- [Optional Semaphore Capabilities](#)
- [Optional Fence Capabilities](#)
- [Timestamp Calibration Capabilities](#)

## 38.1. Additional Image Capabilities

Additional image capabilities, such as larger dimensions or additional sample counts for certain image types, or additional capabilities for *linear tiling* format images, are described in this section.

To query additional capabilities specific to image types, call:

```
VkResult vkGetPhysicalDeviceImageFormatProperties(  
    VkPhysicalDevice           physicalDevice,  
    VkFormat                   format,  
    VkImageType                type,  
    VkImageTiling               tiling,  
    VkImageUsageFlags           usage,  
    VkImageCreateFlags          flags,  
    VkImageFormatProperties*   pImageFormatProperties);
```

- `physicalDevice` is the physical device from which to query the image capabilities.
- `format` is a `VkFormat` value specifying the image format, corresponding to [VkImageCreateInfo::format](#).
- `type` is a `VkImageType` value specifying the image type, corresponding to [VkImageCreateInfo::imageType](#).
- `tiling` is a `VkImageTiling` value specifying the image tiling, corresponding to [VkImageCreateInfo::tiling](#).
- `usage` is a bitmask of `VkImageUsageFlagBits` specifying the intended usage of the image, corresponding to [VkImageCreateInfo::usage](#).
- `flags` is a bitmask of `VkImageCreateFlagBits` specifying additional parameters of the image, corresponding to [VkImageCreateInfo::flags](#).
- `pImageFormatProperties` is a pointer to a `VkImageFormatProperties` structure in which capabilities are returned.

The `format`, `type`, `tiling`, `usage`, and `flags` parameters correspond to parameters that would be consumed by `vkCreateImage` (as members of `VkImageCreateInfo`).

If `format` is not a supported image format, or if the combination of `format`, `type`, `tiling`, `usage`, and `flags` is not supported for images, then `vkGetPhysicalDeviceImageFormatProperties` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`.

The limitations on an image format that are reported by `vkGetPhysicalDeviceImageFormatProperties` have the following property: if `usage1` and `usage2` of type `VkImageUsageFlags` are such that the bits set in `usage1` are a subset of the bits set in `usage2`, and `flags1` and `flags2` of type `VkImageCreateFlags` are such that the bits set in `flags1` are a subset of the bits set in `flags2`, then the limitations for `usage1` and `flags1` **must** be no more strict than the limitations for `usage2` and `flags2`, for all values of `format`, `type`, and `tiling`.

## Valid Usage

- `tiling` **must** not be `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`. (Use `vkGetPhysicalDeviceImageFormatProperties2` instead).

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `format` **must** be a valid `VkFormat` value
- `type` **must** be a valid `VkImageType` value
- `tiling` **must** be a valid `VkImageTiling` value
- `usage` **must** be a valid combination of `VkImageUsageFlagBits` values
- `usage` **must** not be `0`
- `flags` **must** be a valid combination of `VkImageCreateFlagBits` values
- `pImageFormatProperties` **must** be a valid pointer to a `VkImageFormatProperties` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_FORMAT_NOT_SUPPORTED`

The `VkImageFormatProperties` structure is defined as:

```

typedef struct VkImageFormatProperties {
    VkExtent3D          maxExtent;
    uint32_t            maxMipLevels;
    uint32_t            maxArrayLayers;
    VkSampleCountFlags sampleCounts;
    VkDeviceSize        maxResourceSize;
} VkImageFormatProperties;

```

- **maxExtent** are the maximum image dimensions. See the [Allowed Extent Values](#) section below for how these values are constrained by **type**.
- **maxMipLevels** is the maximum number of mipmap levels. **maxMipLevels** **must** be equal to the number of levels in the complete mipmap chain based on the **maxExtent.width**, **maxExtent.height**, and **maxExtent.depth**, except when one of the following conditions is true, in which case it **may** instead be 1:
  - `vkGetPhysicalDeviceImageFormatProperties::tiling` was `VK_IMAGE_TILING_LINEAR`
  - `VkPhysicalDeviceImageFormatInfo2::tiling` was `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`
  - the `VkPhysicalDeviceImageFormatInfo2::pNext` chain included an instance of `VkPhysicalDeviceExternalImageFormatInfo` with a handle type included in the **handleTypes** member for which mipmap image support is not required
  - **image format** is one of those listed in [Formats requiring sampler Y'C<sub>B</sub>C<sub>R</sub> conversion](#) for `VK_IMAGE_ASPECT_COLOR_BIT` [image views](#)
  - **flags** contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`
- **maxArrayLayers** is the maximum number of array layers. **maxArrayLayers** **must** be no less than `VkPhysicalDeviceLimits::maxImageArrayLayers`, except when one of the following conditions is true, in which case it **may** instead be 1:
  - **tiling** is `VK_IMAGE_TILING_LINEAR`
  - **tiling** is `VK_IMAGE_TILING_OPTIMAL` and **type** is `VK_IMAGE_TYPE_3D`
  - **format** is one of those listed in [Formats requiring sampler Y'C<sub>B</sub>C<sub>R</sub> conversion](#) for `VK_IMAGE_ASPECT_COLOR_BIT` [image views](#)
- If **tiling** is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then **maxArrayLayers** **must** not be 0.
- **sampleCounts** is a bitmask of `VkSampleCountFlagBits` specifying all the supported sample counts for this image as described [below](#).
- **maxResourceSize** is an upper bound on the total image size in bytes, inclusive of all image subresources. Implementations **may** have an address space limit on total size of a resource, which is advertised by this property. **maxResourceSize** **must** be at least  $2^{31}$ .

*Note*

There is no mechanism to query the size of an image before creating it, to compare that size against `maxResourceSize`. If an application attempts to create an image that exceeds this limit, the creation will fail and `vkCreateImage` will return `VK_ERROR_OUT_OF_DEVICE_MEMORY`. While the advertised limit **must** be at least  $2^{31}$ , it **may** not be possible to create an image that approaches that size, particularly for `VK_IMAGE_TYPE_1D`.



If the combination of parameters to `vkGetPhysicalDeviceImageFormatProperties` is not supported by the implementation for use in `vkCreateImage`, then all members of `VkImageFormatProperties` will be filled with zero.

*Note*



Filling `VkImageFormatProperties` with zero for unsupported formats is an exception to the usual rule that output structures have undefined contents on error. This exception was unintentional, but is preserved for backwards compatibility.

To determine the image capabilities compatible with an external memory handle type, call:

```
VkResult vkGetPhysicalDeviceExternalImageFormatPropertiesNV(  
    VkPhysicalDevice           physicalDevice,  
    VkFormat                  format,  
    VkImageType               type,  
    VkImageTiling              tiling,  
    VkImageUsageFlags          usage,  
    VkImageCreateFlags         flags,  
    VkExternalMemoryHandleTypeFlagsNV  
    externalHandleType,  
    pExternalImageFormatProperties);
```

- `physicalDevice` is the physical device from which to query the image capabilities
- `format` is the image format, corresponding to `VkImageCreateInfo::format`.
- `type` is the image type, corresponding to `VkImageCreateInfo::imageType`.
- `tiling` is the image tiling, corresponding to `VkImageCreateInfo::tiling`.
- `usage` is the intended usage of the image, corresponding to `VkImageCreateInfo::usage`.
- `flags` is a bitmask describing additional parameters of the image, corresponding to `VkImageCreateInfo::flags`.
- `externalHandleType` is either one of the bits from `VkExternalMemoryHandleTypeFlagBitsNV`, or 0.
- `pExternalImageFormatProperties` is a pointer to a `VkExternalImageFormatPropertiesNV` structure in which capabilities are returned.

If `externalHandleType` is 0, `pExternalImageFormatProperties::imageFormatProperties` will return the same values as a call to `vkGetPhysicalDeviceImageFormatProperties`, and the other members of `ExternalImageFormatProperties` will all be 0. Otherwise, they are filled in as described for

`VkExternalImageFormatPropertiesNV`.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `format` **must** be a valid `VkFormat` value
- `type` **must** be a valid `VkImageType` value
- `tiling` **must** be a valid `VkImageTiling` value
- `usage` **must** be a valid combination of `VkImageUsageFlagBits` values
- `usage` **must** not be `0`
- `flags` **must** be a valid combination of `VkImageCreateFlagBits` values
- `externalHandleType` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values
- `pExternalImageFormatProperties` **must** be a valid pointer to a `VkExternalImageFormatPropertiesNV` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_FORMAT_NOT_SUPPORTED`

The `VkExternalImageFormatPropertiesNV` structure is defined as:

```
typedef struct VkExternalImageFormatPropertiesNV {
    VkImageFormatProperties           imageFormatProperties;
    VkExternalMemoryFeatureFlagsNV   externalMemoryFeatures;
    VkExternalMemoryHandleTypeFlagsNV exportFromImportedHandleTypes;
    VkExternalMemoryHandleTypeFlagsNV compatibleHandleTypes;
} VkExternalImageFormatPropertiesNV;
```

- `imageFormatProperties` will be filled in as when calling `vkGetPhysicalDeviceImageFormatProperties`, but the values returned **may** vary depending on the external handle type requested.
- `externalMemoryFeatures` is a bitmask of `VkExternalMemoryFeatureFlagBitsNV`, indicating properties of the external memory handle type (`vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType`) being queried, or `0` if the external memory handle type is `0`.

- `exportFromImportedHandleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` containing a bit set for every external handle type that **may** be used to create memory from which the handles of the type specified in `vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType` **can** be exported, or 0 if the external memory handle type is 0.
- `compatibleHandleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` containing a bit set for every external handle type that **may** be specified simultaneously with the handle type specified by `vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType` when calling `vkAllocateMemory`, or 0 if the external memory handle type is 0. `compatibleHandleTypes` will always contain `vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType`

Bits which **can** be set in `VkExternalImageFormatPropertiesNV::externalMemoryFeatures`, indicating properties of the external memory handle type, are:

```
typedef enum VkExternalMemoryFeatureFlagBitsNV {
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV = 0x00000001,
    VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_NV = 0x00000002,
    VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_NV = 0x00000004,
    VK_EXTERNAL_MEMORY_FEATURE_FLAG_BITS_MAX_ENUM_NV = 0x7FFFFFFF
} VkExternalMemoryFeatureFlagBitsNV;
```

- `VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV` specifies that external memory of the specified type **must** be created as a dedicated allocation when used in the manner specified.
- `VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_NV` specifies that the implementation supports exporting handles of the specified type.
- `VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_NV` specifies that the implementation supports importing handles of the specified type.

```
typedef VkFlags VkExternalMemoryFeatureFlagsNV;
```

`VkExternalMemoryFeatureFlagsNV` is a bitmask type for setting a mask of zero or more `VkExternalMemoryFeatureFlagBitsNV`.

To query additional capabilities specific to image types, call:

```
VkResult vkGetPhysicalDeviceImageFormatProperties2(
    VkPhysicalDevice                      physicalDevice,
    const VkPhysicalDeviceImageFormatInfo2* pImageFormatInfo,
    VkImageFormatProperties2*              pImageFormatProperties);
```

or the equivalent command

```
VkResult vkGetPhysicalDeviceImageFormatProperties2KHR(  
    VkPhysicalDevice physicalDevice,  
    const VkPhysicalDeviceImageFormatInfo2* pImageFormatInfo,  
    VkImageFormatProperties2* pImageFormatProperties);
```

- `physicalDevice` is the physical device from which to query the image capabilities.
- `pImageFormatInfo` is a pointer to a `VkPhysicalDeviceImageFormatInfo2` structure describing the parameters that would be consumed by `vkCreateImage`.
- `pImageFormatProperties` is a pointer to a `VkImageFormatProperties2` structure in which capabilities are returned.

`vkGetPhysicalDeviceImageFormatProperties2` behaves similarly to `vkGetPhysicalDeviceImageFormatProperties`, with the ability to return extended information in a `pNext` chain of output structures.

## Valid Usage

- If the `pNext` chain of `pImageFormatProperties` contains an instance of `VkAndroidHardwareBufferUsageANDROID`, the `pNext` chain of `pImageFormatInfo` **must** contain an instance of `VkPhysicalDeviceExternalImageFormatInfo` with `handleType` set to `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pImageFormatInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceImageFormatInfo2` structure
- `pImageFormatProperties` **must** be a valid pointer to a `VkImageFormatProperties2` structure

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_FORMAT_NOT_SUPPORTED`

The `VkPhysicalDeviceImageFormatInfo2` structure is defined as:

```

typedef struct VkPhysicalDeviceImageFormatInfo2 {
    VkStructureType      sType;
    const void*        pNext;
    VkFormat             format;
    VkImageType          type;
    VkImageTiling         tiling;
    VkImageUsageFlags     usage;
    VkImageCreateFlags     flags;
} VkPhysicalDeviceImageFormatInfo2;

```

or the equivalent

```
typedef VkPhysicalDeviceImageFormatInfo2 VkPhysicalDeviceImageFormatInfo2KHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure. The **pNext** chain of **VkPhysicalDeviceImageFormatInfo2** is used to provide additional image parameters to **vkGetPhysicalDeviceImageFormatProperties2**.
- **format** is a **VkFormat** value indicating the image format, corresponding to **VkImageCreateInfo::format**.
- **type** is a **VkImageType** value indicating the image type, corresponding to **VkImageCreateInfo::imageType**.
- **tiling** is a **VkImageTiling** value indicating the image tiling, corresponding to **VkImageCreateInfo::tiling**.
- **usage** is a bitmask of **VkImageUsageFlagBits** indicating the intended usage of the image, corresponding to **VkImageCreateInfo::usage**.
- **flags** is a bitmask of **VkImageCreateFlagBits** indicating additional parameters of the image, corresponding to **VkImageCreateInfo::flags**.

The members of **VkPhysicalDeviceImageFormatInfo2** correspond to the arguments to **vkGetPhysicalDeviceImageFormatProperties**, with **sType** and **pNext** added for extensibility.

## Valid Usage

- **tiling** **must** be **VK\_IMAGE\_TILING\_DRM\_FORMAT\_MODIFIER\_EXT** if and only if the **pNext** chain contains **VkPhysicalDeviceDrmFormatModifierInfoEXT**.
- If **tiling** is **VK\_IMAGE\_TILING\_DRM\_FORMAT\_MODIFIER\_EXT** and **flags** contains **VK\_IMAGE\_CREATE\_MUTABLE\_FORMAT\_BIT**, then the **pNext** chain **must** contain **VkImageFormatListCreateInfoKHR** with non-zero **viewFormatCount**.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2`
- Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkImageFormatListCreateInfoKHR`, `VkImageStencilUsageCreateInfoEXT`, `VkPhysicalDeviceExternalImageFormatInfo`, `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`, or `VkPhysicalDeviceImageViewImageFormatInfoEXT`
- Each `sType` member in the `pNext` chain must be unique
- `format` must be a valid `VkFormat` value
- `type` must be a valid `VkImageType` value
- `tiling` must be a valid `VkImageTiling` value
- `usage` must be a valid combination of `VkImageUsageFlagBits` values
- `usage` must not be `0`
- `flags` must be a valid combination of `VkImageCreateFlagBits` values

The `VkImageFormatProperties2` structure is defined as:

```
typedef struct VkImageFormatProperties2 {
    VkStructureType          sType;
    void*                    pNext;
    VkImageFormatProperties  imageFormatProperties;
} VkImageFormatProperties2;
```

or the equivalent

```
typedef VkImageFormatProperties2 VkImageFormatProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure. The `pNext` chain of `VkImageFormatProperties2` is used to allow the specification of additional capabilities to be returned from `vkGetPhysicalDeviceImageFormatProperties2`.
- `imageFormatProperties` is an instance of a `VkImageFormatProperties` structure in which capabilities are returned.

If the combination of parameters to `vkGetPhysicalDeviceImageFormatProperties2` is not supported by the implementation for use in `vkCreateImage`, then all members of `imageFormatProperties` will be filled with zero.

### Note



Filling `imageFormatProperties` with zero for unsupported formats is an exception to the usual rule that output structures have undefined contents on error. This exception was unintentional, but is preserved for backwards compatibility. This exception only applies to `imageFormatProperties`, not `sType`, `pNext`, or any structures chained from `pNext`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2`
- Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of `VkAndroidHardwareBufferUsageANDROID`, `VkExternalImageFormatProperties`, `VkFilterCubicImageViewImageFormatPropertiesEXT`, `VkSamplerYcbcrConversionImageFormatProperties`, or `VkTextureLODGatherFormatPropertiesAMD`
- Each `sType` member in the `pNext` chain **must** be unique

To determine if texture gather functions that take explicit LOD and/or bias argument values can be used with a given image format, add `VkImageFormatProperties2` to the `pNext` chain of the `VkPhysicalDeviceImageFormatInfo2` structure and `VkTextureLODGatherFormatPropertiesAMD` to the `pNext` chain of the `VkImageFormatProperties2` structure.

The `VkTextureLODGatherFormatPropertiesAMD` structure is defined as:

```
typedef struct VkTextureLODGatherFormatPropertiesAMD {  
    VkStructureType    sType;  
    void*              pNext;  
    VkBool32            supportsTextureGatherLODBiasAMD;  
} VkTextureLODGatherFormatPropertiesAMD;
```

- `sType` is the type of this structure.
- `pNext` is `NULL`.
- `supportsTextureGatherLODBiasAMD` tells if the image format can be used with texture gather bias/LOD functions, as introduced by the `VK_AMD_texture_gather_bias_lo`d extension. This field is set by the implementation. User-specified value is ignored.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD`

To determine the image capabilities compatible with an external memory handle type, add `VkPhysicalDeviceExternalImageFormatInfo` to the `pNext` chain of the `VkPhysicalDeviceImageFormatInfo2` structure and `VkExternalImageFormatProperties` to the `pNext`

chain of the `VkImageFormatProperties2` structure.

The `VkPhysicalDeviceExternalImageFormatInfo` structure is defined as:

```
typedef struct VkPhysicalDeviceExternalImageFormatInfo {
    VkStructureType           sType;
    const void*                pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalImageFormatInfo;
```

or the equivalent

```
typedef VkPhysicalDeviceExternalImageFormatInfo
VkPhysicalDeviceExternalImageFormatInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleType` is a `VkExternalMemoryHandleTypeFlagBits` value specifying the memory handle type that will be used with the memory associated with the image.

If `handleType` is `0`, `vkGetPhysicalDeviceImageFormatProperties2` will behave as if `VkPhysicalDeviceExternalImageFormatInfo` was not present, and `VkExternalImageFormatProperties` will be ignored.

If `handleType` is not compatible with the `format`, `type`, `tiling`, `usage`, and `flags` specified in `VkPhysicalDeviceImageFormatInfo2`, then `vkGetPhysicalDeviceImageFormatProperties2` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO`
- If `handleType` is not `0`, `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

Possible values of `VkPhysicalDeviceExternalImageFormatInfo::handleType`, specifying an external memory handle type, are:

```

typedef enum VkExternalMemoryHandleTypeFlagBits {
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT = 0x00000008,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT = 0x00000010,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT = 0x00000020,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT = 0x00000040,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT = 0x00000200,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID = 0x00000400,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT = 0x00000800,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT = 0x00001000,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHR =
VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR =
VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR =
VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHR =
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT_KHR =
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT_KHR =
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT_KHR =
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkExternalMemoryHandleTypeFlagBits;

```

or the equivalent

```
typedef VkExternalMemoryHandleTypeFlagBits VkExternalMemoryHandleTypeFlagBitsKHR;
```

- **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_FD\_BIT** specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls **dup**, **dup2**, **close**, and the non-standard system call **dup3**. Additionally, it **must** be transportable over a socket using an **SCM\_RIGHTS** control message. It owns a reference to the underlying memory resource represented by its Vulkan memory object.
- **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_WIN32\_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the functions **DuplicateHandle**, **CloseHandle**, **CompareObjectHandles**, **GetHandleInformation**, and **SetHandleInformation**. It owns a reference to the underlying memory resource represented by its Vulkan memory object.
- **VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_OPAQUE\_WIN32\_KMT\_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying memory resource represented by its Vulkan memory object, and will therefore become invalid when all Vulkan memory objects

associated with it are destroyed.

- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT` specifies an NT handle returned by `IDXGIResource1::CreateSharedHandle` referring to a Direct3D 10 or 11 texture resource. It owns a reference to the memory used by the Direct3D resource.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT` specifies a global share handle returned by `IDXGIResource::GetSharedHandle` referring to a Direct3D 10 or 11 texture resource. It does not own a reference to the underlying Direct3D resource, and will therefore become invalid when all Vulkan memory objects and Direct3D resources associated with it are destroyed.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT` specifies an NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 heap resource. It owns a reference to the resources used by the Direct3D heap.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT` specifies an NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 committed resource. It owns a reference to the memory used by the Direct3D resource.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT` specifies a host pointer returned by a host memory allocation command. It does not own a reference to the underlying memory resource, and will therefore become invalid if the host memory is freed.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT` specifies a host pointer to *host mapped foreign memory*. It does not own a reference to the underlying memory resource, and will therefore become invalid if the foreign memory is unmapped or otherwise becomes no longer available.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT` is a file descriptor for a Linux `dma_buf`. It owns a reference to the underlying memory resource represented by its Vulkan memory object.
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` specifies an `AHardwareBuffer` object defined by the Android NDK. See [Android Hardware Buffers](#) for more details of this handle type.

Some external memory handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

Table 70. External memory handle types compatibility

Handle type	VkPhysicalDeviceIDProperties::deviceUUID	VkPhysicalDeviceIDProperties::deviceUUID
VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT	Must match	Must match
VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT	Must match	Must match
VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT	Must match	Must match
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT	Must match	Must match
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT	Must match	Must match
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT	Must match	Must match
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT	Must match	Must match
VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT	No restriction	No restriction
VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT	No restriction	No restriction
VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT	No restriction	No restriction
VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID	No restriction	No restriction

*Note*

The above table does not restrict the drivers and devices with which `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT` and `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT` **may** be shared, as these handle types inherently mean memory that does not come from the same device, as they import memory from the host or a foreign device, respectively.

*Note*

Even though the above table does not restrict the drivers and devices with which `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT` **may** be shared, query mechanisms exist in the Vulkan API that prevent the import of incompatible dma-bufs (such as `vkGetMemoryFdPropertiesKHR`) and that prevent incompatible usage of dma-bufs (such as `VkPhysicalDeviceExternalBufferInfo` and `VkPhysicalDeviceExternalImageFormatInfo`).

```
typedef VkFlags VkExternalMemoryHandleTypeFlags;
```

or the equivalent

```
typedef VkExternalMemoryHandleTypeFlags VkExternalMemoryHandleTypeFlagsKHR;
```

`VkExternalMemoryHandleTypeFlags` is a bitmask type for setting a mask of zero or more `VkExternalMemoryHandleTypeFlagBits`.

The `VkExternalImageFormatProperties` structure is defined as:

```
typedef struct VkExternalImageFormatProperties {
    VkStructureType          sType;
    void*                    pNext;
    VkExternalMemoryProperties externalMemoryProperties;
} VkExternalImageFormatProperties;
```

or the equivalent

```
typedef VkExternalImageFormatProperties VkExternalImageFormatPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `externalMemoryProperties` is an instance of the `VkExternalMemoryProperties` structure specifying various capabilities of the external handle type when used with the specified image creation parameters.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES`

The `VkExternalMemoryProperties` structure is defined as:

```
typedef struct VkExternalMemoryProperties {
    VkExternalMemoryFeatureFlags      externalMemoryFeatures;
    VkExternalMemoryHandleTypeFlags   exportFromImportedHandleTypes;
    VkExternalMemoryHandleTypeFlags   compatibleHandleTypes;
} VkExternalMemoryProperties;
```

or the equivalent

```
typedef VkExternalMemoryProperties VkExternalMemoryPropertiesKHR;
```

- `externalMemoryFeatures` is a bitmask of `VkExternalMemoryFeatureFlagBits` specifying the

features of `handleType`.

- `exportFromImportedHandleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying which types of imported handle `handleType` **can** be exported from.
- `compatibleHandleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying handle types which **can** be specified at the same time as `handleType` when creating an image compatible with external memory.

`compatibleHandleTypes` **must** include at least `handleType`. Inclusion of a handle type in `compatibleHandleTypes` does not imply the values returned in `VkImageFormatProperties2` will be the same when `VkPhysicalDeviceExternalImageFormatInfo::handleType` is set to that type. The application is responsible for querying the capabilities of all handle types intended for concurrent use in a single image and intersecting them to obtain the compatible set of capabilities.

Bits which **may** be set in `VkExternalMemoryProperties::externalMemoryFeatures`, specifying features of an external memory handle type, are:

```
typedef enum VkExternalMemoryFeatureFlagBits {
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT = 0x00000001,
    VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT = 0x00000002,
    VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT = 0x00000004,
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_KHR =
VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT,
    VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_KHR =
VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT,
    VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_KHR =
VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT,
    VK_EXTERNAL_MEMORY_FEATURE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkExternalMemoryFeatureFlagBits;
```

or the equivalent

```
typedef VkExternalMemoryFeatureFlagBits VkExternalMemoryFeatureFlagBitsKHR;
```

- `VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT` specifies that images or buffers created with the specified parameters and handle type **must** use the mechanisms defined by `VkMemoryDedicatedRequirements` and `VkMemoryDedicatedAllocateInfo` to create (or import) a dedicated allocation for the image or buffer.
- `VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT` specifies that handles of this type **can** be exported from Vulkan memory objects.
- `VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT` specifies that handles of this type **can** be imported as Vulkan memory objects.

Because their semantics in external APIs roughly align with that of an image or buffer with a dedicated allocation in Vulkan, implementations are **required** to report `VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT` for the following external handle types:

- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`
- `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` for images only

Implementations **must** not report `VK_EXTERNAL_MEMORY_FEATUREDEDICATEDONLYBIT` for buffers with external handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`. Implementations **must** not report `VK_EXTERNAL_MEMORY_FEATUREDEDICATEDONLYBIT` for images or buffers with external handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`.

```
typedef VkFlags VkExternalMemoryFeatureFlags;
```

or the equivalent

```
typedef VkExternalMemoryFeatureFlags VkExternalMemoryFeatureFlagsKHR;
```

`VkExternalMemoryFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalMemoryFeatureFlagBits`.

To query the image capabilities that are compatible with a [Linux DRM format modifier](#), set `VkPhysicalDeviceImageFormatInfo2::tiling` to `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and add `VkPhysicalDeviceDrmFormatModifierInfoEXT` to the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2`.

The `VkPhysicalDeviceDrmFormatModifierInfoEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceImageDrmFormatModifierInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint64_t drmFormatModifier;
    VkSharingMode sharingMode;
    uint32_t queueFamilyIndexCount;
    const uint32_t* pQueueFamilyIndices;
} VkPhysicalDeviceImageDrmFormatModifierInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `drmFormatModifier` is the image's [Linux DRM format modifier](#), corresponding to `VkImageDrmFormatModifierExplicitCreateInfoEXT::modifier` or to `VkImageDrmFormatModifierListCreateInfoEXT::pModifiers`.
- `sharingMode` specifies how the image will be accessed by multiple queue families.
- `queueFamilyIndexCount` is the number of entries in the `pQueueFamilyIndices` array.
- `pQueueFamilyIndices` is a list of queue families that will access the image (ignored if `sharingMode`

is not `VK_SHARING_MODE_CONCURRENT`).

If the `drmFormatModifier` is incompatible with the parameters specified in `VkPhysicalDeviceImageFormatInfo2` and its `pNext` chain, then `vkGetPhysicalDeviceImageFormatProperties2` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`. The implementation **must** support the query of any `drmFormatModifier`, including unknown and invalid modifier values.

## Valid Usage

- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `pQueueFamilyIndices` **must** be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values.
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `queueFamilyIndexCount` **must** be greater than 1.
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` **must** be unique and **must** be less than the `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create `device`.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT`
- `sharingMode` **must** be a valid `VkSharingMode` value

To determine the number of combined image samplers required to support a multi-planar format, add `VkSamplerYcbcrConversionImageFormatProperties` to the `pNext` chain of the `VkImageFormatProperties2` structure in a call to `vkGetPhysicalDeviceImageFormatProperties2`.

The `VkSamplerYcbcrConversionImageFormatProperties` structure is defined as:

```
typedef struct VkSamplerYcbcrConversionImageFormatProperties {
    VkStructureType    sType;
    void*              pNext;
    uint32_t            combinedImageSamplerDescriptorCount;
} VkSamplerYcbcrConversionImageFormatProperties;
```

or the equivalent

```
typedef VkSamplerYcbcrConversionImageFormatProperties
VkSamplerYcbcrConversionImageFormatPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.

- `combinedImageSamplerDescriptorCount` is the number of combined image sampler descriptors that the implementation uses to access the format.

## Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES`

`combinedImageSamplerDescriptorCount` is a number between 1 and the number of planes in the format. A descriptor set layout binding with immutable Y'CbCr conversion samplers will have a maximum `combinedImageSamplerDescriptorCount` which is the maximum across all formats supported by its samplers of the `combinedImageSamplerDescriptorCount` for each format. Descriptor sets with that layout will internally use that maximum `combinedImageSamplerDescriptorCount` descriptors for each descriptor in the binding. This expanded number of descriptors will be consumed from the descriptor pool when a descriptor set is allocated, and counts towards the `maxDescriptorSetSamplers`, `maxDescriptorSetSampledImages`, `maxPerStageDescriptorSamplers`, and `maxPerStageDescriptorSampledImages` limits.

### Note

All descriptors in a binding use the same maximum `combinedImageSamplerDescriptorCount` descriptors to allow implementations to use a uniform stride for dynamic indexing of the descriptors in the binding.



For example, consider a descriptor set layout binding with two descriptors and immutable samplers for multi-planar formats that have `VkSamplerYcbcrConversionImageFormatProperties::combinedImageSamplerDescriptorCount` values of 2 and 3 respectively. There are two descriptors in the binding and the maximum `combinedImageSamplerDescriptorCount` is 3, so descriptor sets with this layout consume 6 descriptors from the descriptor pool. To create a descriptor pool that allows allocating four descriptor sets with this layout, `descriptorCount` must be at least 24.

To obtain optimal Android hardware buffer usage flags for specific image creation parameters, attach an instance of `VkAndroidHardwareBufferUsageANDROID` to the `pNext` chain of a `VkImageFormatProperties2` structure passed to `vkGetPhysicalDeviceImageFormatProperties2`. This structure is defined as:

```
typedef struct VkAndroidHardwareBufferUsageANDROID {
    VkStructureType sType;
    void* pNext;
    uint64_t androidHardwareBufferUsage;
} VkAndroidHardwareBufferUsageANDROID;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `androidHardwareBufferUsage` returns the Android hardware buffer usage flags.

The `androidHardwareBufferUsage` field **must** include Android hardware buffer usage flags listed in the [AHardwareBuffer Usage Equivalence](#) table when the corresponding Vulkan image usage or image creation flags are included in the `usage` or `flags` fields of `VkPhysicalDeviceImageFormatInfo2`. It **must** include at least one GPU usage flag (`AHARDWAREBUFFER_USAGE_GPU_*`), even if none of the corresponding Vulkan usages or flags are requested.

*Note*



Requiring at least one GPU usage flag ensures that Android hardware buffer memory will be allocated in a memory pool accessible to the Vulkan implementation, and that specializing the memory layout based on usage flags does not prevent it from being compatible with Vulkan. Implementations **may** avoid unnecessary restrictions caused by this requirement by using vendor usage flags to indicate that only the Vulkan uses indicated in `VkImageFormatProperties2` are required.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_USAGE_ANDROID`

To determine if cubic filtering can be used with a given image format and a given image view type add `VkPhysicalDeviceImageViewImageFormatInfoEXT` to the `pNext` chain of the `VkPhysicalDeviceImageFormatInfo2` structure and `VkFilterCubicImageViewImageFormatPropertiesEXT` to the `pNext` chain of the `VkImageFormatProperties2` structure.

The `VkPhysicalDeviceImageViewImageFormatInfoEXT` structure is defined as:

```
typedef struct VkPhysicalDeviceImageViewImageFormatInfoEXT {
    VkStructureType sType;
    void* pNext;
    VkImageViewType imageViewType;
} VkPhysicalDeviceImageViewImageFormatInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `imageViewType` is a `VkImageViewType` value specifying the type of the image view.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_VIEW_IMAGE_FORMAT_INFO_EXT`
- `imageViewType` **must** be a valid `VkImageViewType` value

The `VkFilterCubicImageViewImageFormatPropertiesEXT` structure is defined as:

```

typedef struct VkFilterCubicImageViewImageFormatPropertiesEXT {
    VkStructureType    sType;
    void*             pNext;
    VkBool32           filterCubic;
    VkBool32           filterCubicMinmax ;
} VkFilterCubicImageViewImageFormatPropertiesEXT;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **filterCubic** tells if image format, image type and image view type **can** be used with cubic filtering. This field is set by the implementation. User-specified value is ignored.
- **filterCubicMinmax** tells if image format, image type and image view type **can** be used with cubic filtering and minmax filtering. This field is set by the implementation. User-specified value is ignored.

## Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_FILTER_CUBIC_IMAGE_VIEW_IMAGE_FORMAT_PROPERTIES_EXT`

## Valid Usage

- If the **pNext** chain of the `VkImageFormatProperties2` structure contains an instance of `VkFilterCubicImageViewImageFormatPropertiesEXT`, the **pNext** chain of the `VkPhysicalDeviceImageFormatInfo2` structure **must** contain an instance of `VkPhysicalDeviceImageViewImageFormatInfoEXT` with an **imageViewType** that is compatible with **imageType**.

### 38.1.1. Supported Sample Counts

`vkGetPhysicalDeviceImageFormatProperties` returns a bitmask of `VkSampleCountFlagBits` in **sampleCounts** specifying the supported sample counts for the image parameters.

**sampleCounts** will be set to `VK_SAMPLE_COUNT_1_BIT` if at least one of the following conditions is true:

- **tiling** is `VK_IMAGE_TILING_LINEAR`
- **type** is not `VK_IMAGE_TYPE_2D`
- **flags** contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`
- Neither the `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` flag nor the `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT` flag in `VkFormatProperties::optimalTilingFeatures` returned by `vkGetPhysicalDeviceFormatProperties` is set
- `VkPhysicalDeviceExternalImageFormatInfo::handleType` is an external handle type for which multisampled image support is not required.

- `format` is one of those listed in Formats requiring sampler YC<sub>B</sub>C<sub>R</sub> conversion for `VK_IMAGE_ASPECT_COLOR_BIT` image views
- `usage` contains `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`
- `usage` contains `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`

Otherwise, the bits set in `sampleCounts` will be the sample counts supported for the specified values of `usage` and `format`. For each bit set in `usage`, the supported sample counts relate to the limits in `VkPhysicalDeviceLimits` as follows:

- If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` and `format` is a floating- or fixed-point color format, a superset of `VkPhysicalDeviceLimits::framebufferColorSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `format` includes a depth aspect, a superset of `VkPhysicalDeviceLimits::framebufferDepthSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, and `format` includes a stencil aspect, a superset of `VkPhysicalDeviceLimits::framebufferStencilSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` includes a color aspect, a superset of `VkPhysicalDeviceLimits::sampledImageColorSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` includes a depth aspect, a superset of `VkPhysicalDeviceLimits::sampledImageDepthSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_SAMPLED_BIT`, and `format` is an integer format, a superset of `VkPhysicalDeviceLimits::sampledImageIntegerSampleCounts`
- If `usage` includes `VK_IMAGE_USAGE_STORAGE_BIT`, a superset of `VkPhysicalDeviceLimits::storageImageSampleCounts`

If multiple bits are set in `usage`, `sampleCounts` will be the intersection of the per-usage values described above.

If none of the bits described above are set in `usage`, then there is no corresponding limit in `VkPhysicalDeviceLimits`. In this case, `sampleCounts` **must** include at least `VK_SAMPLE_COUNT_1_BIT`.

### 38.1.2. Allowed Extent Values Based On Image Type

Implementations **may** support extent values larger than the required minimum/maximum values for certain types of images subject to the constraints below.

*Note*



Implementations **must** support images with dimensions up to the required minimum/maximum values for all types of images. It follows that the query for additional capabilities **must** return extent values that are at least as large as the required values.

For `VK_IMAGE_TYPE_1D`:

- `maxExtent.width`  $\geq$  `VkPhysicalDeviceLimits.maxImageDimension1D`
- `maxExtent.height` = 1

- `maxExtent.depth` = 1

For `VK_IMAGE_TYPE_2D` when `flags` does not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

- `maxExtent.width` ≥ `VkPhysicalDeviceLimits.maxImageDimension2D`
- `maxExtent.height` ≥ `VkPhysicalDeviceLimits.maxImageDimension2D`
- `maxExtent.depth` = 1

For `VK_IMAGE_TYPE_2D` when `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

- `maxExtent.width` ≥ `VkPhysicalDeviceLimits.maxImageDimensionCube`
- `maxExtent.height` ≥ `VkPhysicalDeviceLimits.maxImageDimensionCube`
- `maxExtent.depth` = 1

For `VK_IMAGE_TYPE_3D`:

- `maxExtent.width` ≥ `VkPhysicalDeviceLimits.maxImageDimension3D`
- `maxExtent.height` ≥ `VkPhysicalDeviceLimits.maxImageDimension3D`
- `maxExtent.depth` ≥ `VkPhysicalDeviceLimits.maxImageDimension3D`

## 38.2. Additional Buffer Capabilities

To query the external handle types supported by buffers, call:

```
void vkGetPhysicalDeviceExternalBufferProperties(
    VkPhysicalDevice                  physicalDevice,
    const VkPhysicalDeviceExternalBufferInfo* pExternalBufferInfo,
    VkExternalBufferProperties*        pExternalBufferProperties);
```

or the equivalent command

```
void vkGetPhysicalDeviceExternalBufferPropertiesKHR(
    VkPhysicalDevice                  physicalDevice,
    const VkPhysicalDeviceExternalBufferInfo* pExternalBufferInfo,
    VkExternalBufferProperties*        pExternalBufferProperties);
```

- `physicalDevice` is the physical device from which to query the buffer capabilities.
- `pExternalBufferInfo` is a pointer to a `VkPhysicalDeviceExternalBufferInfo` structure describing the parameters that would be consumed by `vkCreateBuffer`.
- `pExternalBufferProperties` is a pointer to a `VkExternalBufferProperties` structure in which capabilities are returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pExternalBufferInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceExternalBufferInfo` structure
- `pExternalBufferProperties` **must** be a valid pointer to a `VkExternalBufferProperties` structure

The `VkPhysicalDeviceExternalBufferInfo` structure is defined as:

```
typedef struct VkPhysicalDeviceExternalBufferInfo {  
    VkStructureType           sType;  
    const void*               pNext;  
    VkBufferCreateFlags       flags;  
    VkBufferUsageFlags        usage;  
    VkExternalMemoryHandleTypeFlagBits handleType;  
} VkPhysicalDeviceExternalBufferInfo;
```

or the equivalent

```
typedef VkPhysicalDeviceExternalBufferInfo VkPhysicalDeviceExternalBufferInfoKHR;
```

- `sType` is the type of this structure
- `pNext` is NULL or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkBufferCreateFlagBits` describing additional parameters of the buffer, corresponding to `VkBufferCreateInfo::flags`.
- `usage` is a bitmask of `VkBufferUsageFlagBits` describing the intended usage of the buffer, corresponding to `VkBufferCreateInfo::usage`.
- `handleType` is a `VkExternalMemoryHandleTypeFlagBits` value specifying the memory handle type that will be used with the memory associated with the buffer.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of `VkBufferCreateFlagBits` values
- `usage` **must** be a valid combination of `VkBufferUsageFlagBits` values
- `usage` **must** not be `0`
- `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

The `VkExternalBufferProperties` structure is defined as:

```
typedef struct VkExternalBufferProperties {
    VkStructureType          sType;
    void*                    pNext;
    VkExternalMemoryProperties externalMemoryProperties;
} VkExternalBufferProperties;
```

or the equivalent

```
typedef VkExternalBufferProperties VkExternalBufferPropertiesKHR;
```

- `sType` is the type of this structure
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `externalMemoryProperties` is an instance of the `VkExternalMemoryProperties` structure specifying various capabilities of the external handle type when used with the specified buffer creation parameters.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES`
- `pNext` must be `NULL`

## 38.3. Optional Semaphore Capabilities

Semaphores **may** support import and export of their `payload` to external handles. To query the external handle types supported by semaphores, call:

```
void vkGetPhysicalDeviceExternalSemaphoreProperties(
    VkPhysicalDevice           physicalDevice,
    const VkPhysicalDeviceExternalSemaphoreInfo* pExternalSemaphoreInfo,
    VkExternalSemaphoreProperties* pExternalSemaphoreProperties);
```

or the equivalent command

```
void vkGetPhysicalDeviceExternalSemaphorePropertiesKHR(
    VkPhysicalDevice           physicalDevice,
    const VkPhysicalDeviceExternalSemaphoreInfo* pExternalSemaphoreInfo,
    VkExternalSemaphoreProperties* pExternalSemaphoreProperties);
```

- `physicalDevice` is the physical device from which to query the semaphore capabilities.
- `pExternalSemaphoreInfo` is a pointer to a `VkPhysicalDeviceExternalSemaphoreInfo` structure

describing the parameters that would be consumed by `vkCreateSemaphore`.

- `pExternalSemaphoreProperties` is a pointer to a `VkExternalSemaphoreProperties` structure in which capabilities are returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pExternalSemaphoreInfo` **must** be a valid pointer to a `VkPhysicalDeviceExternalSemaphoreInfo` structure
- `pExternalSemaphoreProperties` **must** be a valid pointer to a `VkExternalSemaphoreProperties` structure

The `VkPhysicalDeviceExternalSemaphoreInfo` structure is defined as:

```
typedef struct VkPhysicalDeviceExternalSemaphoreInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalSemaphoreInfo;
```

or the equivalent

```
typedef VkPhysicalDeviceExternalSemaphoreInfo
VkPhysicalDeviceExternalSemaphoreInfoKHR;
```

- `sType` is the type of this structure
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleType` is a `VkExternalSemaphoreHandleTypeFlagBits` value specifying the external semaphore handle type for which capabilities will be returned.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of `VkSemaphoreTypeCreateInfoKHR`
- `handleType` **must** be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

Bits which **may** be set in `VkPhysicalDeviceExternalSemaphoreInfo::handleType`, specifying an external semaphore handle type, are:

```

typedef enum VkExternalSemaphoreHandleTypeFlagBits {
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT = 0x00000008,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT = 0x00000010,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT_KHR =
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR =
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR =
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT_KHR =
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT_KHR =
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkExternalSemaphoreHandleTypeFlagBits;

```

or the equivalent

```

typedef VkExternalSemaphoreHandleTypeFlagBits
VkExternalSemaphoreHandleTypeFlagBitsKHR;

```

- **VK\_EXTERNAL\_SEMAPHORE\_HANDLE\_TYPE\_OPAQUE\_FD\_BIT** specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls **dup**, **dup2**, **close**, and the non-standard system call **dup3**. Additionally, it **must** be transportable over a socket using an **SCM\_RIGHTS** control message. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.
- **VK\_EXTERNAL\_SEMAPHORE\_HANDLE\_TYPE\_OPAQUE\_WIN32\_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the functions **DuplicateHandle**, **CloseHandle**, **CompareObjectHandles**, **GetHandleInformation**, and **SetHandleInformation**. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.
- **VK\_EXTERNAL\_SEMAPHORE\_HANDLE\_TYPE\_OPAQUE\_WIN32\_KMT\_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive represented by its Vulkan semaphore object, and will therefore become invalid when all Vulkan semaphore objects associated with it are destroyed.
- **VK\_EXTERNAL\_SEMAPHORE\_HANDLE\_TYPE\_D3D12\_FENCE\_BIT** specifies an NT handle returned by **ID3D12Device::CreateSharedHandle** referring to a Direct3D 12 fence. It owns a reference to the underlying synchronization primitive associated with the Direct3D fence.
- **VK\_EXTERNAL\_SEMAPHORE\_HANDLE\_TYPE\_SYNC\_FD\_BIT** specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence object. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive

associated with the file descriptor. Implementations which support importing this handle type **must** accept any type of sync or fence FD supported by the native system they are running on.

*Note*

Handles of type `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT` generated by the implementation may represent either Linux Sync Files or Android Fences at the implementation's discretion. Applications **should** only use operations defined for both types of file descriptors, unless they know via means external to Vulkan the type of the file descriptor, or are prepared to deal with the system-defined operation failures resulting from using the wrong type.



Some external semaphore handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

Table 71. External semaphore handle types compatibility

Handle type	VkPhysicalDeviceIDProperties::deviceUUID	VkPhysicalDeviceIDProperties::deviceUUID
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT	Must match	Must match
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT	Must match	Must match
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT	Must match	Must match
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT	Must match	Must match
VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT	No restriction	No restriction

```
typedef VkFlags VkExternalSemaphoreHandleTypeFlags;
```

or the equivalent

```
typedef VkExternalSemaphoreHandleTypeFlags VkExternalSemaphoreHandleTypeFlagsKHR;
```

`VkExternalSemaphoreHandleTypeFlags` is a bitmask type for setting a mask of zero or more `VkExternalSemaphoreHandleTypeFlagBits`.

The `VkExternalSemaphoreProperties` structure is defined as:

```
typedef struct VkExternalSemaphoreProperties {
    VkStructureType           sType;
    void*                     pNext;
    VkExternalSemaphoreHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalSemaphoreHandleTypeFlags compatibleHandleTypes;
    VkExternalSemaphoreFeatureFlags   externalSemaphoreFeatures;
} VkExternalSemaphoreProperties;
```

or the equivalent

```
typedef VkExternalSemaphoreProperties VkExternalSemaphorePropertiesKHR;
```

- `exportFromImportedHandleTypes` is a bitmask of `VkExternalSemaphoreHandleTypeFlagBits` specifying which types of imported handle `handleType` **can** be exported from.
- `compatibleHandleTypes` is a bitmask of `VkExternalSemaphoreHandleTypeFlagBits` specifying handle types which **can** be specified at the same time as `handleType` when creating a semaphore.
- `externalSemaphoreFeatures` is a bitmask of `VkExternalSemaphoreFeatureFlagBits` describing the

features of `handleType`.

If `handleType` is not supported by the implementation, then `VkExternalSemaphoreProperties::externalSemaphoreFeatures` will be set to zero.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES`
- `pNext` **must** be `NULL`

Possible values of `VkExternalSemaphoreProperties::externalSemaphoreFeatures`, specifying the features of an external semaphore handle type, are:

```
typedef enum VkExternalSemaphoreFeatureFlagBits {
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT = 0x00000002,
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT_KHR =
VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT,
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT_KHR =
VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT,
    VK_EXTERNAL_SEMAPHORE_FEATURE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkExternalSemaphoreFeatureFlagBits;
```

or the equivalent

```
typedef VkExternalSemaphoreFeatureFlagBits VkExternalSemaphoreFeatureFlagBitsKHR;
```

- `VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT` specifies that handles of this type **can** be exported from Vulkan semaphore objects.
- `VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT` specifies that handles of this type **can** be imported as Vulkan semaphore objects.

```
typedef VkFlags VkExternalSemaphoreFeatureFlags;
```

or the equivalent

```
typedef VkExternalSemaphoreFeatureFlags VkExternalSemaphoreFeatureFlagsKHR;
```

`VkExternalSemaphoreFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalSemaphoreFeatureFlagBits`.

## 38.4. Optional Fence Capabilities

Fences **may** support import and export of their [payload](#) to external handles. To query the external handle types supported by fences, call:

```
void vkGetPhysicalDeviceExternalFenceProperties(  
    VkPhysicalDevice physicalDevice,  
    const VkPhysicalDeviceExternalFenceInfo* pExternalFenceInfo,  
    VkExternalFenceProperties* pExternalFenceProperties);
```

or the equivalent command

```
void vkGetPhysicalDeviceExternalFencePropertiesKHR(  
    VkPhysicalDevice physicalDevice,  
    const VkPhysicalDeviceExternalFenceInfo* pExternalFenceInfo,  
    VkExternalFenceProperties* pExternalFenceProperties);
```

- `physicalDevice` is the physical device from which to query the fence capabilities.
- `pExternalFenceInfo` is a pointer to a `VkPhysicalDeviceExternalFenceInfo` structure describing the parameters that would be consumed by `vkCreateFence`.
- `pExternalFenceProperties` is a pointer to a `VkExternalFenceProperties` structure in which capabilities are returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pExternalFenceInfo` **must** be a valid pointer to a valid `VkPhysicalDeviceExternalFenceInfo` structure
- `pExternalFenceProperties` **must** be a valid pointer to a `VkExternalFenceProperties` structure

The `VkPhysicalDeviceExternalFenceInfo` structure is defined as:

```
typedef struct VkPhysicalDeviceExternalFenceInfo {  
    VkStructureType sType;  
    const void* pNext;  
    VkExternalFenceHandleTypeFlagBits handleType;  
} VkPhysicalDeviceExternalFenceInfo;
```

or the equivalent

```
typedef VkPhysicalDeviceExternalFenceInfo VkPhysicalDeviceExternalFenceInfoKHR;
```

- `sType` is the type of this structure
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleType` is a `VkExternalFenceHandleTypeFlagBits` value indicating an external fence handle type for which capabilities will be returned.

*Note*

Handles of type `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` generated by the implementation may represent either Linux Sync Files or Android Fences at the implementation's discretion. Applications **should** only use operations defined for both types of file descriptors, unless they know via means external to Vulkan the type of the file descriptor, or are prepared to deal with the system-defined operation failures resulting from using the wrong type.



## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO`
- `pNext` **must** be `NULL`
- `handleType` **must** be a valid `VkExternalFenceHandleTypeFlagBits` value

Bits which **may** be set in `VkPhysicalDeviceExternalFenceInfo::handleType`, and in the `exportFromImportedHandleTypes` and `compatibleHandleTypes` members of `VkExternalFenceProperties`, to indicate external fence handle types, are:

```
typedef enum VkExternalFenceHandleTypeFlagBits {
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT = 0x00000008,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT_KHR =
VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR =
VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR =
VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT_KHR =
VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT,
    VK_EXTERNAL_FENCE_HANDLE_TYPE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkExternalFenceHandleTypeFlagBits;
```

or the equivalent

```
typedef VkExternalFenceHandleTypeFlagBits VkExternalFenceHandleTypeFlagBitsKHR;
```

- `VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT` specifies a POSIX file descriptor handle that has

only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls `dup`, `dup2`, `close`, and the non-standard system call `dup3`. Additionally, it **must** be transportable over a socket using an `SCM_RIGHTS` control message. It owns a reference to the underlying synchronization primitive represented by its Vulkan fence object.

- `VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT` specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying synchronization primitive represented by its Vulkan fence object.
- `VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT` specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive represented by its Vulkan fence object, and will therefore become invalid when all Vulkan fence objects associated with it are destroyed.
- `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive associated with the file descriptor. Implementations which support importing this handle type **must** accept any type of sync or fence FD supported by the native system they are running on.

Some external fence handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

Table 72. External fence handle types compatibility

Handle type	<code>VkPhysicalDeviceIDProperties::deviceUUID</code>	<code>VkPhysicalDeviceIDProperties::deviceUUID</code>
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT</code>	Must match	Must match
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT</code>	Must match	Must match
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</code>	Must match	Must match
<code>VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT</code>	No restriction	No restriction

```
typedef VkFlags VkExternalFenceHandleTypeFlags;
```

or the equivalent

```
typedef VkExternalFenceHandleTypeFlags VkExternalFenceHandleTypeFlagsKHR;
```

`VkExternalFenceHandleTypeFlags` is a bitmask type for setting a mask of zero or more `VkExternalFenceHandleTypeFlagBits`.

The `VkExternalFenceProperties` structure is defined as:

```
typedef struct VkExternalFenceProperties {
    VkStructureType           sType;
    void*                     pNext;
    VkExternalFenceHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalFenceHandleTypeFlags compatibleHandleTypes;
    VkExternalFenceFeatureFlags externalFenceFeatures;
} VkExternalFenceProperties;
```

or the equivalent

```
typedef VkExternalFenceProperties VkExternalFencePropertiesKHR;
```

- `exportFromImportedHandleTypes` is a bitmask of `VkExternalFenceHandleTypeFlagBits` indicating which types of imported handle `handleType` **can** be exported from.
- `compatibleHandleTypes` is a bitmask of `VkExternalFenceHandleTypeFlagBits` specifying handle types which **can** be specified at the same time as `handleType` when creating a fence.
- `externalFenceFeatures` is a bitmask of `VkExternalFenceFeatureFlagBits` indicating the features of `handleType`.

If `handleType` is not supported by the implementation, then `VkExternalFenceProperties::externalFenceFeatures` will be set to zero.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES`
- `pNext` must be `NULL`

Bits which **may** be set in `VkExternalFenceProperties::externalFenceFeatures`, indicating features of a fence external handle type, are:

```
typedef enum VkExternalFenceFeatureFlagBits {
    VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT = 0x00000002,
    VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT_KHR =
VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT,
    VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT_KHR =
VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT,
    VK_EXTERNAL_FENCE_FEATURE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkExternalFenceFeatureFlagBits;
```

or the equivalent

```
typedef VkExternalFenceFeatureFlagBits VkExternalFenceFeatureFlagBitsKHR;
```

- `VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT` specifies handles of this type **can** be exported from Vulkan fence objects.
- `VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT` specifies handles of this type **can** be imported to Vulkan fence objects.

```
typedef VkFlags VkExternalFenceFeatureFlags;
```

or the equivalent

```
typedef VkExternalFenceFeatureFlags VkExternalFenceFeatureFlagsKHR;
```

`VkExternalFenceFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalFenceFeatureFlagBits`.

## 38.5. Timestamp Calibration Capabilities

To query the set of time domains for which a physical device supports timestamp calibration, call:

```
VkResult vkGetPhysicalDeviceCalibrateableTimeDomainsEXT(  
    VkPhysicalDevice physicalDevice,  
    uint32_t* pTimeDomainCount,  
    VkTimeDomainEXT* pTimeDomains);
```

- `physicalDevice` is the physical device from which to query the set of calibrateable time domains.
- `pTimeDomainCount` is a pointer to an integer related to the number of calibrateable time domains available or queried, as described below.
- `pTimeDomains` is either `NULL` or a pointer to an array of `VkTimeDomainEXT` values, indicating the supported calibrateable time domains.

If `pTimeDomains` is `NULL`, then the number of calibrateable time domains supported for the given `physicalDevice` is returned in `pTimeDomainCount`. Otherwise, `pTimeDomainCount` **must** point to a variable set by the user to the number of elements in the `pTimeDomains` array, and on return the variable is overwritten with the number of values actually written to `pTimeDomains`. If the value of `pTimeDomainCount` is less than the number of calibrateable time domains supported, at most `pTimeDomainCount` values will be written to `pTimeDomains`. If `pTimeDomainCount` is smaller than the number of calibrateable time domains supported for the given `physicalDevice`, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS` to indicate that not all the available values were returned.

## Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pTimeDomainCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pTimeDomainCount` is not `0`, and `pTimeDomains` is not `NULL`, `pTimeDomains` **must** be a valid pointer to an array of `pTimeDomainCount` `VkTimeDomainEXT` values

## Return Codes

### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

# Chapter 39. Debugging

To aid developers in tracking down errors in the application's use of Vulkan, particularly in combination with an external debugger or profiler, *debugging extensions* may be available.

The [VkObjectType](#) enumeration defines values, each of which corresponds to a specific Vulkan handle type. These values **can** be used to associate debug information with a particular type of object through one or more extensions.

```

typedef enum VkObjectType {
    VK_OBJECT_TYPE_UNKNOWN = 0,
    VK_OBJECT_TYPE_INSTANCE = 1,
    VK_OBJECT_TYPE_PHYSICAL_DEVICE = 2,
    VK_OBJECT_TYPE_DEVICE = 3,
    VK_OBJECT_TYPE_QUEUE = 4,
    VK_OBJECT_TYPE_SEMAPHORE = 5,
    VK_OBJECT_TYPE_COMMAND_BUFFER = 6,
    VK_OBJECT_TYPE_FENCE = 7,
    VK_OBJECT_TYPE_DEVICE_MEMORY = 8,
    VK_OBJECT_TYPE_BUFFER = 9,
    VK_OBJECT_TYPE_IMAGE = 10,
    VK_OBJECT_TYPE_EVENT = 11,
    VK_OBJECT_TYPE_QUERY_POOL = 12,
    VK_OBJECT_TYPE_BUFFER_VIEW = 13,
    VK_OBJECT_TYPE_IMAGE_VIEW = 14,
    VK_OBJECT_TYPE_SHADER_MODULE = 15,
    VK_OBJECT_TYPE_PIPELINE_CACHE = 16,
    VK_OBJECT_TYPE_PIPELINE_LAYOUT = 17,
    VK_OBJECT_TYPE_RENDER_PASS = 18,
    VK_OBJECT_TYPE_PIPELINE = 19,
    VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT = 20,
    VK_OBJECT_TYPE_SAMPLER = 21,
    VK_OBJECT_TYPE_DESCRIPTOR_POOL = 22,
    VK_OBJECT_TYPE_DESCRIPTOR_SET = 23,
    VK_OBJECT_TYPE_FRAMEBUFFER = 24,
    VK_OBJECT_TYPE_COMMAND_POOL = 25,
    VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION = 1000156000,
    VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE = 1000085000,
    VK_OBJECT_TYPE_SURFACE_KHR = 1000000000,
    VK_OBJECT_TYPE_SWAPCHAIN_KHR = 1000001000,
    VK_OBJECT_TYPE_DISPLAY_KHR = 1000002000,
    VK_OBJECT_TYPE_DISPLAY_MODE_KHR = 1000002001,
    VK_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT = 1000011000,
    VK_OBJECT_TYPE_OBJECT_TABLE_NVX = 1000086000,
    VK_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NVX = 1000086001,
    VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT = 1000128000,
    VK_OBJECT_TYPE_VALIDATION_CACHE_EXT = 1000160000,
    VK_OBJECT_TYPE_ACCELERATION_STRUCTURE_NV = 1000165000,
    VK_OBJECT_TYPE_PERFORMANCE_CONFIGURATION_INTEL = 1000210000,
    VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR =
VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE,
    VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_KHR =
VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION,
    VK_OBJECT_TYPE_MAX_ENUM = 0x7FFFFFFF
} VkObjectType;

```

Table 73. *VkObjectType* and Vulkan Handle Relationship

<b>VkObjectType</b>	<b>Vulkan Handle Type</b>
VK_OBJECT_TYPE_UNKNOWN	Unknown/Undefined Handle
VK_OBJECT_TYPE_INSTANCE	VkInstance
VK_OBJECT_TYPE_PHYSICAL_DEVICE	VkPhysicalDevice
VK_OBJECT_TYPE_DEVICE	VkDevice
VK_OBJECT_TYPE_QUEUE	VkQueue
VK_OBJECT_TYPE_SEMAPHORE	VkSemaphore
VK_OBJECT_TYPE_COMMAND_BUFFER	VkCommandBuffer
VK_OBJECT_TYPE_FENCE	VkFence
VK_OBJECT_TYPE_DEVICE_MEMORY	VkDeviceMemory
VK_OBJECT_TYPE_BUFFER	VkBuffer
VK_OBJECT_TYPE_IMAGE	VkImage
VK_OBJECT_TYPE_EVENT	VkEvent
VK_OBJECT_TYPE_QUERY_POOL	VkQueryPool
VK_OBJECT_TYPE_BUFFER_VIEW	VkBufferView
VK_OBJECT_TYPE_IMAGE_VIEW	VkImageView
VK_OBJECT_TYPE_SHADER_MODULE	VkShaderModule
VK_OBJECT_TYPE_PIPELINE_CACHE	VkPipelineCache
VK_OBJECT_TYPE_PIPELINE_LAYOUT	VkPipelineLayout
VK_OBJECT_TYPE_RENDER_PASS	VkRenderPass
VK_OBJECT_TYPE_PIPELINE	VkPipeline
VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT	VkDescriptorSetLayout
VK_OBJECT_TYPE_SAMPLER	VkSampler
VK_OBJECT_TYPE_DESCRIPTOR_POOL	VkDescriptorPool
VK_OBJECT_TYPE_DESCRIPTOR_SET	VkDescriptorSet
VK_OBJECT_TYPE_FRAMEBUFFER	VkFramebuffer
VK_OBJECT_TYPE_COMMAND_POOL	VkCommandPool
VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION	VkSamplerYcbcrConversion
VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE	VkDescriptorUpdateTemplate
VK_OBJECT_TYPE_SURFACE_KHR	VkSurfaceKHR
VK_OBJECT_TYPE_SWAPCHAIN_KHR	VkSwapchainKHR
VK_OBJECT_TYPE_DISPLAY_KHR	VkDisplayKHR
VK_OBJECT_TYPE_DISPLAY_MODE_KHR	VkDisplayModeKHR
VK_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT	VkDebugReportCallbackEXT
VK_OBJECT_TYPE_OBJECT_TABLE_NVX	VkObjectTableNVX
VK_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NVX	VkIndirectCommandsLayoutNVX

VkObjectType	Vulkan Handle Type
VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT	VkDebugUtilsMessengerEXT
VK_OBJECT_TYPE_VALIDATION_CACHE_EXT	VkValidationCacheEXT
VK_OBJECT_TYPE_ACCELERATION_STRUCTURE_NV	VkAccelerationStructureNV
VK_OBJECT_TYPE_PERFORMANCE_CONFIGURATION_INTEL	VkPerformanceConfigurationINTEL

If this Specification was generated with any such extensions included, they will be described in the remainder of this chapter.

## 39.1. Debug Utilities

Vulkan provides flexible debugging utilities for debugging an application.

The [Object Debug Annotation](#) section describes how to associate either a name or binary data with a specific Vulkan object.

The [Queue Labels](#) section describes how to annotate and group the work submitted to a queue.

The [Command Buffer Labels](#) section describes how to associate logical elements of the scene with commands in a [VkCommandBuffer](#).

The [Debug Messengers](#) section describes how to create debug messenger objects associated with an application supplied callback to capture debug messages from a variety of Vulkan components.

### 39.1.1. Object Debug Annotation

It can be useful for an application to provide its own content relative to a specific Vulkan object. The following commands allow application developers to associate user-defined information with Vulkan objects.

#### Object Naming

An object can be provided a user-defined name by calling [vkSetDebugUtilsObjectNameEXT](#) as defined below.

```
VkResult vkSetDebugUtilsObjectNameEXT(
    VkDevice device,
    const VkDebugUtilsObjectNameInfoEXT* pNameInfo);
```

- `device` is the device that created the object.
- `pNameInfo` is a pointer to a [VkDebugUtilsObjectNameInfoEXT](#) structure specifying parameters of the name to set on the object.

## Valid Usage

- `pNameInfo->objectType` **must** not be `VK_OBJECT_TYPE_UNKNOWN`
- `pNameInfo->objectHandle` **must** not be `VK_NULL_HANDLE`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pNameInfo` **must** be a valid pointer to a valid `VkDebugUtilsObjectNameInfoEXT` structure

## Host Synchronization

- Host access to `pNameInfo.objectHandle` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDebugUtilsObjectNameInfoEXT` structure is defined as:

```
typedef struct VkDebugUtilsObjectNameInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkObjectType objectType;
    uint64_t objectHandle;
    const char* pObjectName;
} VkDebugUtilsObjectNameInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `objectType` is a `VkObjectType` specifying the type of the object to be named.
- `objectHandle` is the object to be named.
- `pObjectName` is a null-terminated UTF-8 string specifying the name to apply to `objectHandle`.

Applications **may** change the name associated with an object simply by calling `vkSetDebugUtilsObjectNameEXT` again with a new string. If `pObjectName` is an empty string, then any

previously set name is removed.

## Valid Usage

- If `objectType` is `VK_OBJECT_TYPE_UNKNOWN`, `objectHandle` **must** not be `VK_NULL_HANDLE`
- If `objectType` is not `VK_OBJECT_TYPE_UNKNOWN`, `objectHandle` **must** be `VK_NULL_HANDLE` or a valid Vulkan handle of the type associated with `objectType` as defined in the [VkObjectType and Vulkan Handle Relationship](#) table

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT`
- `pNext` **must** be `NULL`
- `objectType` **must** be a valid `VkObjectType` value
- If `pObjectName` is not `NULL`, `pObjectName` **must** be a null-terminated UTF-8 string

## Object Data Association

In addition to setting a name for an object, debugging and validation layers **may** have uses for additional binary data on a per-object basis that have no other place in the Vulkan API.

For example, a `VkShaderModule` could have additional debugging data attached to it to aid in offline shader tracing.

Additional data can be attached to an object by calling `vkSetDebugUtilsObjectTagEXT` as defined below.

```
VkResult vkSetDebugUtilsObjectTagEXT(  
    VkDevice                                     device,  
    const VkDebugUtilsObjectTagInfoEXT* pTagInfo);
```

- `device` is the device that created the object.
- `pTagInfo` is a pointer to a `VkDebugUtilsObjectTagInfoEXT` structure specifying parameters of the tag to attach to the object.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pTagInfo` **must** be a valid pointer to a valid `VkDebugUtilsObjectTagInfoEXT` structure

## Host Synchronization

- Host access to `pTagInfo.objectHandle` must be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDebugUtilsObjectTagInfoEXT` structure is defined as:

```
typedef struct VkDebugUtilsObjectTagInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkObjectType       objectType;
    uint64_t           objectHandle;
    uint64_t           tagName;
    size_t              tagSize;
    const void*        pTag;
} VkDebugUtilsObjectTagInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `objectType` is a `VkObjectType` specifying the type of the object to be named.
- `objectHandle` is the object to be tagged.
- `tagName` is a numerical identifier of the tag.
- `tagSize` is the number of bytes of data to attach to the object.
- `pTag` is a pointer to an array of `tagSize` bytes containing the data to be associated with the object.

The `tagName` parameter gives a name or identifier to the type of data being tagged. This can be used by debugging layers to easily filter for only data that can be used by that implementation.

## Valid Usage

- `objectType` must not be `VK_OBJECT_TYPE_UNKNOWN`
- `objectHandle` must be a valid Vulkan handle of the type associated with `objectType` as defined in the `VkObjectType` and `Vulkan Handle Relationship` table

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT`
- `pNext` **must** be `NULL`
- `objectType` **must** be a valid `VkObjectType` value
- `pTag` **must** be a valid pointer to an array of `tagSize` bytes
- `tagSize` **must** be greater than `0`

### 39.1.2. Queue Labels

All Vulkan work must be submitted using queues. It is possible for an application to use multiple queues, each containing multiple command buffers, when performing work. It can be useful to identify which queue, or even where in a queue, something has occurred.

To begin identifying a region using a debug label inside a queue, you may use the `vkQueueBeginDebugUtilsLabelEXT` command.

Then, when the region of interest has passed, you may end the label region using `vkQueueEndDebugUtilsLabelEXT`.

Additionally, a single debug label may be inserted at any time using `vkQueueInsertDebugUtilsLabelEXT`.

A queue debug label region is opened by calling:

```
void vkQueueBeginDebugUtilsLabelEXT(  
    VkQueue                      queue,  
    const VkDebugUtilsLabelEXT* pLabelInfo);
```

- `queue` is the queue in which to start a debug label region.
- `pLabelInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label region to open.

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- `pLabelInfo` **must** be a valid pointer to a valid `VkDebugUtilsLabelEXT` structure

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	Any	-

The `VkDebugUtilsLabelEXT` structure is defined as:

```
typedef struct VkDebugUtilsLabelEXT {
    VkStructureType    sType;
    const void*        pNext;
    const char*        pLabelName;
    float              color[4];
} VkDebugUtilsLabelEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pLabelName` is a pointer to a null-terminated UTF-8 string containing the name of the label.
- `color` is an optional RGBA color value that can be associated with the label. A particular implementation `may` choose to ignore this color value. The values contain RGBA values in order, in the range 0.0 to 1.0. If all elements in `color` are set to 0.0 then it is ignored.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT`
- `pNext` **must** be `NULL`
- `pLabelName` **must** be a null-terminated UTF-8 string

A queue debug label region is closed by calling:

```
void vkQueueEndDebugUtilsLabelEXT(
    VkQueue                queue);
```

- `queue` is the queue in which a debug label region should be closed.

The calls to `vkQueueBeginDebugUtilsLabelEXT` and `vkQueueEndDebugUtilsLabelEXT` **must** be matched and balanced.

## Valid Usage

- There **must** be an outstanding `vkQueueBeginDebugUtilsLabelEXT` command prior to the `vkQueueEndDebugUtilsLabelEXT` on the queue

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	Any	-

A single label can be inserted into a queue by calling:

```
void vkQueueInsertDebugUtilsLabelEXT(  
    VkQueue  
    const VkDebugUtilsLabelEXT*  
                                queue,  
                                pLabelInfo);
```

- `queue` is the queue into which a debug label will be inserted.
- `pLabelInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label to insert.

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- `pLabelInfo` **must** be a valid pointer to a valid `VkDebugUtilsLabelEXT` structure

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
-	-	Any	-

### 39.1.3. Command Buffer Labels

Typical Vulkan applications will submit many command buffers in each frame, with each command buffer containing a large number of individual commands. Being able to logically

annotate regions of command buffers that belong together as well as hierarchically subdivide the frame is important to a developer's ability to navigate the commands viewed holistically.

To identify the beginning of a debug label region in a command buffer, [vkCmdBeginDebugUtilsLabelEXT](#) can be used as defined below.

To indicate the end of a debug label region in a command buffer, [vkCmdEndDebugUtilsLabelEXT](#) can be used.

To insert a single command buffer debug label inside of a command buffer, [vkCmdInsertDebugUtilsLabelEXT](#) can be used as defined below.

A command buffer debug label region can be opened by calling:

```
void vkCmdBeginDebugUtilsLabelEXT(  
    VkCommandBuffer  
    const VkDebugUtilsLabelEXT*  
                                commandBuffer,  
                                pLabelInfo);
```

- **commandBuffer** is the command buffer into which the command is recorded.
- **pLabelInfo** is a pointer to a [VkDebugUtilsLabelEXT](#) structure specifying parameters of the label region to open.

### Valid Usage (Implicit)

- **commandBuffer** **must** be a valid [VkCommandBuffer](#) handle
- **pLabelInfo** **must** be a valid pointer to a valid [VkDebugUtilsLabelEXT](#) structure
- **commandBuffer** **must** be in the [recording state](#)
- The [VkCommandPool](#) that **commandBuffer** was allocated from **must** support graphics, or compute operations

### Host Synchronization

- Host access to the [VkCommandPool](#) that **commandBuffer** was allocated from **must** be externally synchronized

### Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

A command buffer label region can be closed by calling:

```
void vkCmdEndDebugUtilsLabelEXT(  
    VkCommandBuffer  
                                commandBuffer);
```

- `commandBuffer` is the command buffer into which the command is recorded.

An application **may** open a debug label region in one command buffer and close it in another, or otherwise split debug label regions across multiple command buffers or multiple queue submissions. When viewed from the linear series of submissions to a single queue, the calls to `vkCmdBeginDebugUtilsLabelEXT` and `vkCmdEndDebugUtilsLabelEXT` **must** be matched and balanced.

## Valid Usage

- There **must** be an outstanding `vkCmdBeginDebugUtilsLabelEXT` command prior to the `vkCmdEndDebugUtilsLabelEXT` on the queue that `commandBuffer` is submitted to
- If `commandBuffer` is a secondary command buffer, there **must** be an outstanding `vkCmdBeginDebugUtilsLabelEXT` command recorded to `commandBuffer` that has not previously been ended by a call to `vkCmdEndDebugUtilsLabelEXT`.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

A single debug label can be inserted into a command buffer by calling:

```
void vkCmdInsertDebugUtilsLabelEXT(
    VkCommandBuffer
    const VkDebugUtilsLabelEXT* commandBuffer,
    pLabelInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label to insert.

### Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pLabelInfo` **must** be a valid pointer to a valid `VkDebugUtilsLabelEXT` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

### Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

### Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

#### 39.1.4. Debug Messengers

Vulkan allows an application to register multiple callbacks with any Vulkan component wishing to report debug information. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. A primary producer of callback messages are the validation layers. An application **can** register callbacks even when no validation layers are enabled, but they will only be called for the Vulkan loader and, if implemented, other layer and driver events.

A `VkDebugUtilsMessengerEXT` is a messenger object which handles passing along debug messages to a provided debug callback.

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDebugUtilsMessengerEXT)
```

The debug messenger will provide detailed feedback on the application's use of Vulkan when events of interest occur. When an event of interest does occur, the debug messenger will submit a debug message to the debug callback that was provided during its creation. Additionally, the debug messenger is responsible with filtering out debug messages that the callback is not interested in and will only provide desired debug messages.

A debug messenger triggers a debug callback with a debug message when an event of interest occurs. To create a debug messenger which will trigger a debug callback, call:

```
VkResult vkCreateDebugUtilsMessengerEXT(  
    VkInstance instance,  
    const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkDebugUtilsMessengerEXT* pMessenger);
```

- **instance** the instance the messenger will be used with.
- **pCreateInfo** is a pointer to a [VkDebugUtilsMessengerCreateInfoEXT](#) structure containing the callback pointer, as well as defining conditions under which this messenger will trigger the callback.
- **pAllocator** controls host memory allocation as described in the [Memory Allocation](#) chapter.
- **pMessenger** is a pointer to a [VkDebugUtilsMessengerEXT](#) handle in which the created object is returned.

## Valid Usage (Implicit)

- **instance must** be a valid [VkInstance](#) handle
- **pCreateInfo must** be a valid pointer to a valid [VkDebugUtilsMessengerCreateInfoEXT](#) structure
- If **pAllocator** is not **NULL**, **pAllocator must** be a valid pointer to a valid [VkAllocationCallbacks](#) structure
- **pMessenger must** be a valid pointer to a [VkDebugUtilsMessengerEXT](#) handle

## Return Codes

### Success

- [VK\\_SUCCESS](#)

### Failure

- [VK\\_ERROR\\_OUT\\_OF\\_HOST\\_MEMORY](#)

The application **must** ensure that [vkCreateDebugUtilsMessengerEXT](#) is not executed in parallel with any Vulkan command that is also called with **instance** or child of **instance** as the dispatchable argument.

The definition of `VkDebugUtilsMessengerCreateInfoEXT` is:

```
typedef struct VkDebugUtilsMessengerCreateInfoEXT {
    VkStructureType          sType;
    const void*               pNext;
    VkDebugUtilsMessengerCreateFlagsEXT   flags;
    VkDebugUtilsMessageSeverityFlagsEXT   messageSeverity;
    VkDebugUtilsMessageTypeFlagsEXT       messageType;
    PFN_vkDebugUtilsMessengerCallbackEXT pfnUserCallback;
    void*                         pUserData;
} VkDebugUtilsMessengerCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is 0 and reserved for future use.
- `messageSeverity` is a bitmask of `VkDebugUtilsMessageSeverityFlagBitsEXT` specifying which severity of event(s) will cause this callback to be called.
- `messageType` is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) will cause this callback to be called.
- `pfnUserCallback` is the application callback function to call.
- `pUserData` is user data to be passed to the callback.

For each `VkDebugUtilsMessengerEXT` that is created the `VkDebugUtilsMessengerCreateInfoEXT::messageSeverity` and `VkDebugUtilsMessengerCreateInfoEXT::messageType` determine when that `VkDebugUtilsMessengerCreateInfoEXT::pfnUserCallback` is called. The process to determine if the user's `pfnUserCallback` is triggered when an event occurs is as follows:

1. The implementation will perform a bitwise AND of the event's `VkDebugUtilsMessageSeverityFlagBitsEXT` with the `messageSeverity` provided during creation of the `VkDebugUtilsMessengerEXT` object.
  - a. If the value is 0, the message is skipped.
2. The implementation will perform bitwise AND of the event's `VkDebugUtilsMessageTypeFlagBitsEXT` with the `messageType` provided during the creation of the `VkDebugUtilsMessengerEXT` object.
  - a. If the value is 0, the message is skipped.
3. The callback will trigger a debug message for the current event

The callback will come directly from the component that detected the event, unless some other layer intercepts the calls for its own purposes (filter them in a different way, log to a system error log, etc.).

An application **can** receive multiple callbacks if multiple `VkDebugUtilsMessengerEXT` objects are created. A callback will always be executed in the same thread as the originating Vulkan call.

A callback **can** be called from multiple threads simultaneously (if the application is making Vulkan

calls from multiple threads).

## Valid Usage

- `pfnUserCallback` **must** be a valid `PFN_vkDebugUtilsMessengerCallbackEXT`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT`
- `flags` **must** be `0`
- `messageSeverity` **must** be a valid combination of `VkDebugUtilsMessageSeverityFlagBitsEXT` values
- `messageSeverity` **must** not be `0`
- `messageType` **must** be a valid combination of `VkDebugUtilsMessageTypeFlagBitsEXT` values
- `messageType` **must** not be `0`
- `pfnUserCallback` **must** be a valid `PFN_vkDebugUtilsMessengerCallbackEXT` value

Bits which **can** be set in `VkDebugUtilsMessengerCreateInfoEXT::messageSeverity`, specifying event severities which cause a debug messenger to call the callback, are:

```
typedef enum VkDebugUtilsMessageSeverityFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT = 0x00000001,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT = 0x00000010,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT = 0x00000100,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT = 0x00001000,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDebugUtilsMessageSeverityFlagBitsEXT;
```

- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT` specifies the most verbose output indicating all diagnostic messages from the Vulkan loader, layers, and drivers should be captured.
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT` specifies an informational message such as resource details that may be handy when debugging an application.
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT` specifies use of Vulkan that **may** expose an app bug. Such cases may not be immediately harmful, such as a fragment shader outputting to a location with no attachment. Other cases **may** point to behavior that is almost certainly bad when unintended such as using an image whose memory has not been filled. In general if you see a warning but you know that the behavior is intended/desired, then simply ignore the warning.
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT` specifies that the application has violated a valid usage condition of the specification.

*Note*

The values of `VkDebugUtilsMessageSeverityFlagBitsEXT` are sorted based on severity. The higher the flag value, the more severe the message. This allows for simple boolean operation comparisons when looking at `VkDebugUtilsMessageSeverityFlagBitsEXT` values.

For example:



```
if (messageSeverity >=
VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
    // Do something for warnings and errors
}
```

In addition, space has been left between the enums to allow for later addition of new severities in between the existing values.

```
typedef VkFlags VkDebugUtilsMessageSeverityFlagsEXT;
```

`VkDebugUtilsMessageSeverityFlagsEXT` is a bitmask type for setting a mask of zero or more `VkDebugUtilsMessageSeverityFlagBitsEXT`.

Bits which **can** be set in `VkDebugUtilsMessengerCreateInfoEXT::messageType`, specifying event types which cause a debug messenger to call the callback, are:

```
typedef enum VkDebugUtilsMessageTypeFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT = 0x00000001,
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT = 0x00000002,
    VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT = 0x00000004,
    VK_DEBUG_UTILS_MESSAGE_TYPE_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDebugUtilsMessageTypeFlagBitsEXT;
```

- `VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT` specifies that some general event has occurred. This is typically a non-specification, non-performance event.
- `VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT` specifies that something has occurred during validation against the Vulkan specification that may indicate invalid behavior.
- `VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT` specifies a potentially non-optimal use of Vulkan, e.g. using `vkCmdClearColorImage` when setting `VkAttachmentDescription::loadOp` to `VK_ATTACHMENT_LOAD_OP_CLEAR` would have worked.

```
typedef VkFlags VkDebugUtilsMessageTypeFlagsEXT;
```

`VkDebugUtilsMessageTypeFlagsEXT` is a bitmask type for setting a mask of zero or more `VkDebugUtilsMessageTypeFlagBitsEXT`.

The prototype for the `VkDebugUtilsMessengerCreateInfoEXT::pfnUserCallback` function implemented by the application is:

```
typedef VkBool32 (VKAPI_PTR *PFN_vkDebugUtilsMessengerCallbackEXT)(  
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,  
    VkDebugUtilsMessageTypeFlagsEXT messageTypes,  
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,  
    void* pUserData);
```

- `messageSeverity` specifies the `VkDebugUtilsMessageSeverityFlagBitsEXT` that triggered this callback.
- `messageTypes` is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) triggered this callback.
- `pCallbackData` contains all the callback related data in the `VkDebugUtilsMessengerCallbackDataEXT` structure.
- `pUserData` is the user data provided when the `VkDebugUtilsMessengerEXT` was created.

The callback **must** not call `vkDestroyDebugUtilsMessengerEXT`.

The callback returns a `VkBool32`, which is interpreted in a layer-specified manner. The application **should** always return `VK_FALSE`. The `VK_TRUE` value is reserved for use in layer development.

The definition of `VkDebugUtilsMessengerCallbackDataEXT` is:

```
typedef struct VkDebugUtilsMessengerCallbackDataEXT {  
    VkStructureType sType;  
    const void* pNext;  
    VkDebugUtilsMessengerCallbackDataFlagsEXT flags;  
    const char* pMessageIdName;  
    int32_t messageIdNumber;  
    const char* pMessage;  
    uint32_t queueLabelCount;  
    const VkDebugUtilsLabelEXT* pQueueLabels;  
    uint32_t cmdBufLabelCount;  
    const VkDebugUtilsLabelEXT* pCmdBufLabels;  
    uint32_t objectCount;  
    const VkDebugUtilsObjectNameInfoEXT* pObjects;  
} VkDebugUtilsMessengerCallbackDataEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is 0 and reserved for future use.
- `pMessageIdName` is a null-terminated string that identifies the particular message ID that is associated with the provided message. If the message corresponds to a validation layer message, then this string may contain the portion of the Vulkan specification that is believed to have been

violated.

- `messageIdNumber` is the ID number of the triggering message. If the message corresponds to a validation layer message, then this number is related to the internal number associated with the message being triggered.
- `pMessage` is a null-terminated string detailing the trigger conditions.
- `queueLabelCount` is a count of items contained in the `pQueueLabels` array.
- `pQueueLabels` is NULL or a pointer to an array of `VkDebugUtilsLabelEXT` active in the current `VkQueue` at the time the callback was triggered. Refer to [Queue Labels](#) for more information.
- `cmdBufLabelCount` is a count of items contained in the `pCmdBufLabels` array.
- `pCmdBufLabels` is NULL or a pointer to an array of `VkDebugUtilsLabelEXT` active in the current `VkCommandBuffer` at the time the callback was triggered. Refer to [Command Buffer Labels](#) for more information.
- `objectCount` is a count of items contained in the `pObjects` array.
- `pObjects` is a pointer to an array of `VkDebugUtilsObjectNameInfoEXT` objects related to the detected issue. The array is roughly in order of importance, but the 0th element is always guaranteed to be the most important object for this message.

*Note*



This structure should only be considered valid during the lifetime of the triggered callback.

Since adding queue and command buffer labels behaves like pushing and popping onto a stack, the order of both `pQueueLabels` and `pCmdBufLabels` is based on the order the labels were defined. The result is that the first label in either `pQueueLabels` or `pCmdBufLabels` will be the first defined (and therefore the oldest) while the last label in each list will be the most recent.

*Note*

`pQueueLabels` will only be non-NULL if one of the objects in `pObjects` can be related directly to a defined `VkQueue` which has had one or more labels associated with it.

Likewise, `pCmdBufLabels` will only be non-NULL if one of the objects in `pObjects` can be related directly to a defined `VkCommandBuffer` which has had one or more labels associated with it. Additionally, while command buffer labels allow for beginning and ending across different command buffers, the debug messaging framework **cannot** guarantee that labels in `pCmdBufLabels` will contain those defined outside of the associated command buffer. This is partially due to the fact that the association of one command buffer with another may not have been defined at the time the debug message is triggered.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- If `pMessageIdName` is not `NULL`, `pMessageIdName` **must** be a null-terminated UTF-8 string
- `pMessage` **must** be a null-terminated UTF-8 string
- If `queueLabelCount` is not `0`, `pQueueLabels` **must** be a valid pointer to an array of `queueLabelCount` valid `VkDebugUtilsLabelEXT` structures
- If `cmdBufLabelCount` is not `0`, `pCmdBufLabels` **must** be a valid pointer to an array of `cmdBufLabelCount` valid `VkDebugUtilsLabelEXT` structures
- If `objectCount` is not `0`, `pObjects` **must** be a valid pointer to an array of `objectCount` valid `VkDebugUtilsObjectNameInfoEXT` structures

There may be times that a user wishes to intentionally submit a debug message. To do this, call:

```
void vkSubmitDebugUtilsMessageEXT(  
    VkInstance instance,  
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,  
    VkDebugUtilsMessageTypeFlagsEXT messageTypes,  
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData);
```

- `instance` is the debug stream's `VkInstance`.
- `messageSeverity` is the `VkDebugUtilsMessageSeverityFlagBitsEXT` severity of this event/message.
- `messageTypes` is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) to identify with this message.
- `pCallbackData` contains all the callback related data in the `VkDebugUtilsMessengerCallbackDataEXT` structure.

The call will propagate through the layers and generate callback(s) as indicated by the message's flags. The parameters are passed on to the callback in addition to the `pUserData` value that was defined at the time the messenger was registered.

## Valid Usage

- `objectType` member of each element of `pCallbackData->pObjects` **must** not be `VK_OBJECT_TYPE_UNKNOWN`

## Valid Usage (Implicit)

- **instance** **must** be a valid `VkInstance` handle
- **messageSeverity** **must** be a valid `VkDebugUtilsMessageSeverityFlagBitsEXT` value
- **messageTypes** **must** be a valid combination of `VkDebugUtilsMessageTypeFlagBitsEXT` values
- **messageTypes** **must** not be `0`
- **pCallbackData** **must** be a valid pointer to a valid `VkDebugUtilsMessengerCallbackDataEXT` structure

To destroy a `VkDebugUtilsMessengerEXT` object, call:

```
void vkDestroyDebugUtilsMessengerEXT(  
    VkInstance                      instance,  
    VkDebugUtilsMessengerEXT        messenger,  
    const VkAllocationCallbacks* pAllocator);
```

- **instance** the instance where the callback was created.
- **messenger** the `VkDebugUtilsMessengerEXT` object to destroy. **messenger** is an externally synchronized object and **must** not be used on more than one thread at a time. This means that `vkDestroyDebugUtilsMessengerEXT` **must** not be called when a callback is active.
- **pAllocator** controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- If `VkAllocationCallbacks` were provided when **messenger** was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when **messenger** was created, **pAllocator** **must** be `NULL`

## Valid Usage (Implicit)

- **instance** **must** be a valid `VkInstance` handle
- **messenger** **must** be a valid `VkDebugUtilsMessengerEXT` handle
- If **pAllocator** is not `NULL`, **pAllocator** **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **messenger** **must** have been created, allocated, or retrieved from **instance**

## Host Synchronization

- Host access to `messenger` **must** be externally synchronized

The application **must** ensure that `vkDestroyDebugUtilsMessengerEXT` is not executed in parallel with any Vulkan command that is also called with `instance` or child of `instance` as the dispatchable argument.

## 39.2. Debug Markers

Debug markers provide a flexible way for debugging and validation layers to receive annotation and debug information.

The [Object Annotation](#) section describes how to associate a name or binary data with a Vulkan object.

The [Command Buffer Markers](#) section describes how to associate logical elements of the scene with commands in the command buffer.

### 39.2.1. Object Annotation

The commands in this section allow application developers to associate user-defined information with Vulkan objects at will.

An object can be given a user-friendly name by calling:

```
VkResult vkDebugMarkerSetObjectNameEXT(  
    VkDevice device,  
    const VkDebugMarkerObjectNameInfoEXT* pNameInfo);
```

- `device` is the device that created the object.
- `pNameInfo` is a pointer to a `VkDebugMarkerObjectNameInfoEXT` structure specifying the parameters of the name to set on the object.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pNameInfo` **must** be a valid pointer to a valid `VkDebugMarkerObjectNameInfoEXT` structure

### Host Synchronization

- Host access to `pNameInfo.object` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDebugMarkerObjectNameInfoEXT` structure is defined as:

```
typedef struct VkDebugMarkerObjectNameInfoEXT {
    VkStructureType           sType;
    const void*                pNext;
    VkDebugReportObjectTypeEXT objectType;
    uint64_t                  object;
    const char*                pObjectName;
} VkDebugMarkerObjectNameInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `objectType` is a `VkDebugReportObjectTypeEXT` specifying the type of the object to be named.
- `object` is the object to be named.
- `pObjectName` is a null-terminated UTF-8 string specifying the name to apply to `object`.

Applications **may** change the name associated with an object simply by calling `vkDebugMarkerSetObjectNameEXT` again with a new string. To remove a previously set name, `pObjectName` **should** be set to an empty string.

## Valid Usage

- `objectType` **must** not be `VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT`
- `object` **must** not be `VK_NULL_HANDLE`
- `object` **must** be a Vulkan object of the type associated with `objectType` as defined in `VkDebugReportObjectTypeEXT` and Vulkan Handle Relationship.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_NAME_INFO_EXT`
- `pNext` **must** be `NULL`
- `objectType` **must** be a valid `VkDebugReportObjectTypeEXT` value
- `pObjectName` **must** be a null-terminated UTF-8 string

In addition to setting a name for an object, debugging and validation layers may have uses for additional binary data on a per-object basis that has no other place in the Vulkan API. For example, a `VkShaderModule` could have additional debugging data attached to it to aid in offline shader tracing. To attach data to an object, call:

```
VkResult vkDebugMarkerSetObjectTagEXT(  
    VkDevice device,  
    const VkDebugMarkerObjectTagInfoEXT* pTagInfo);
```

- `device` is the device that created the object.
- `pTagInfo` is a pointer to a `VkDebugMarkerObjectTagInfoEXT` structure specifying the parameters of the tag to attach to the object.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pTagInfo` **must** be a valid pointer to a valid `VkDebugMarkerObjectTagInfoEXT` structure

## Host Synchronization

- Host access to `pTagInfo.object` **must** be externally synchronized

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDebugMarkerObjectTagInfoEXT` structure is defined as:

```

typedef struct VkDebugMarkerObjectTagInfoEXT {
    VkStructureType           sType;
    const void*              pNext;
    VkDebugReportObjectTypeEXT objectType;
    uint64_t                  object;
    uint64_t                  tagName;
    size_t                     tagSize;
    const void*              pTag;
} VkDebugMarkerObjectTagInfoEXT;

```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to an extension-specific structure.
- **objectType** is a [VkDebugReportObjectTypeEXT](#) specifying the type of the object to be named.
- **object** is the object to be tagged.
- **tagName** is a numerical identifier of the tag.
- **tagSize** is the number of bytes of data to attach to the object.
- **pTag** is a pointer to an array of **tagSize** bytes containing the data to be associated with the object.

The **tagName** parameter gives a name or identifier to the type of data being tagged. This can be used by debugging layers to easily filter for only data that can be used by that implementation.

### Valid Usage

- **objectType** **must** not be [VK\\_DEBUG\\_REPORT\\_OBJECT\\_TYPE\\_UNKNOWN\\_EXT](#)
- **object** **must** not be [VK\\_NULL\\_HANDLE](#)
- **object** **must** be a Vulkan object of the type associated with **objectType** as defined in [VkDebugReportObjectTypeEXT](#) and [Vulkan Handle Relationship](#).

### Valid Usage (Implicit)

- **sType** **must** be [VK\\_STRUCTURE\\_TYPE\\_DEBUG\\_MARKER\\_OBJECT\\_TAG\\_INFO\\_EXT](#)
- **pNext** **must** be **NULL**
- **objectType** **must** be a valid [VkDebugReportObjectTypeEXT](#) value
- **pTag** **must** be a valid pointer to an array of **tagSize** bytes
- **tagSize** **must** be greater than **0**

## 39.2.2. Command Buffer Markers

Typical Vulkan applications will submit many command buffers in each frame, with each command buffer containing a large number of individual commands. Being able to logically annotate regions of command buffers that belong together as well as hierarchically subdivide the

frame is important to a developer's ability to navigate the commands viewed holistically.

The marker commands `vkCmdDebugMarkerBeginEXT` and `vkCmdDebugMarkerEndEXT` define regions of a series of commands that are grouped together, and they can be nested to create a hierarchy. The `vkCmdDebugMarkerInsertEXT` command allows insertion of a single label within a command buffer.

A marker region can be opened by calling:

```
void vkCmdDebugMarkerBeginEXT(  
    VkCommandBuffer  
    const VkDebugMarkerMarkerInfoEXT*  
                                commandBuffer,  
                                pMarkerInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pMarkerInfo` is a pointer to a `VkDebugMarkerMarkerInfoEXT` structure specifying the parameters of the marker region to open.

### Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pMarkerInfo` **must** be a valid pointer to a valid `VkDebugMarkerMarkerInfoEXT` structure
- `commandBuffer` **must** be in the `recording` state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

### Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

### Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

The `VkDebugMarkerMarkerInfoEXT` structure is defined as:

```
typedef struct VkDebugMarkerMarkerInfoEXT {
    VkStructureType sType;
    const void* pNext;
    const char* pMarkerName;
    float color[4];
} VkDebugMarkerMarkerInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pMarkerName` is a pointer to a null-terminated UTF-8 string containing the name of the marker.
- `color` is an **optional** RGBA color value that can be associated with the marker. A particular implementation **may** choose to ignore this color value. The values contain RGBA values in order, in the range 0.0 to 1.0. If all elements in `color` are set to 0.0 then it is ignored.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT`
- `pNext` **must** be `NULL`
- `pMarkerName` **must** be a null-terminated UTF-8 string

A marker region can be closed by calling:

```
void vkCmdDebugMarkerEndEXT(
    VkCommandBuffer commandBuffer);
```

- `commandBuffer` is the command buffer into which the command is recorded.

An application **may** open a marker region in one command buffer and close it in another, or otherwise split marker regions across multiple command buffers or multiple queue submissions. When viewed from the linear series of submissions to a single queue, the calls to `vkCmdDebugMarkerBeginEXT` and `vkCmdDebugMarkerEndEXT` **must** be matched and balanced.

## Valid Usage

- There **must** be an outstanding `vkCmdDebugMarkerBeginEXT` command prior to the `vkCmdDebugMarkerEndEXT` on the queue that `commandBuffer` is submitted to
- If `commandBuffer` is a secondary command buffer, there **must** be an outstanding `vkCmdDebugMarkerBeginEXT` command recorded to `commandBuffer` that has not previously been ended by a call to `vkCmdDebugMarkerEndEXT`.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

A single marker label can be inserted into a command buffer by calling:

```
void vkCmdDebugMarkerInsertEXT(  
    VkCommandBuffer  
    const VkDebugMarkerMarkerInfoEXT*  
                                commandBuffer,  
                                pMarkerInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pMarkerInfo` is a pointer to a `VkDebugMarkerMarkerInfoEXT` structure specifying the parameters of the marker to insert.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pMarkerInfo` **must** be a valid pointer to a valid `VkDebugMarkerMarkerInfoEXT` structure
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute	

### 39.3. Debug Report Callbacks

Debug report callbacks are represented by `VkDebugReportCallbackEXT` handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDebugReportCallbackEXT)
```

Debug report callbacks give more detailed feedback on the application's use of Vulkan when events of interest occur.

To register a debug report callback, an application uses `vkCreateDebugReportCallbackEXT`.

```
VkResult vkCreateDebugReportCallbackEXT(  
    VkInstance instance,  
    const VkDebugReportCallbackCreateInfoEXT* pCreateInfo,  
    const VkAllocationCallbacks* pAllocator,  
    VkDebugReportCallbackEXT* pCallback);
```

- `instance` the instance the callback will be logged on.
- `pCreateInfo` is a pointer to a `VkDebugReportCallbackCreateInfoEXT` structure defining the conditions under which this callback will be called.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.
- `pCallback` is a pointer to a `VkDebugReportCallbackEXT` handle in which the created object is returned.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkDebugReportCallbackCreateInfoEXT` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pCallback` **must** be a valid pointer to a `VkDebugReportCallbackEXT` handle

## Return Codes

### Success

- `VK_SUCCESS`

### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`

The definition of `VkDebugReportCallbackCreateInfoEXT` is:

```
typedef struct VkDebugReportCallbackCreateInfoEXT {
    VkStructureType          sType;
    const void*              pNext;
    VkDebugReportFlagsEXT    flags;
    PFN_vkDebugReportCallbackEXT pfnCallback;
    void*                    pUserData;
} VkDebugReportCallbackCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of `VkDebugReportFlagBitsEXT` specifying which event(s) will cause this callback to be called.
- `pfnCallback` is the application callback function to call.
- `pUserData` is user data to be passed to the callback.

For each `VkDebugReportCallbackEXT` that is created the `VkDebugReportCallbackCreateInfoEXT::flags` determine when that `VkDebugReportCallbackCreateInfoEXT::pfnCallback` is called. When an event happens, the implementation will do a bitwise AND of the event's `VkDebugReportFlagBitsEXT` flags to each `VkDebugReportCallbackEXT` object's flags. For each non-zero result the corresponding callback will be called. The callback will come directly from the component that detected the event, unless some other layer intercepts the calls for its own purposes (filter them in a different way, log to a system error log, etc.).

An application **may** receive multiple callbacks if multiple `VkDebugReportCallbackEXT` objects were

created. A callback will always be executed in the same thread as the originating Vulkan call.

A callback may be called from multiple threads simultaneously (if the application is making Vulkan calls from multiple threads).

## Valid Usage (Implicit)

- **sType** **must** be `VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT`
- **flags** **must** be a valid combination of `VkDebugReportFlagBitsEXT` values
- **pfnCallback** **must** be a valid `PFN_vkDebugReportCallbackEXT` value

Bits which **can** be set in `VkDebugReportCallbackCreateInfoEXT::flags`, specifying events which cause a debug report, are:

```
typedef enum VkDebugReportFlagBitsEXT {
    VK_DEBUG_REPORT_INFORMATION_BIT_EXT = 0x00000001,
    VK_DEBUG_REPORT_WARNING_BIT_EXT = 0x00000002,
    VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT = 0x00000004,
    VK_DEBUG_REPORT_ERROR_BIT_EXT = 0x00000008,
    VK_DEBUG_REPORT_DEBUG_BIT_EXT = 0x00000010,
    VK_DEBUG_REPORT_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDebugReportFlagBitsEXT;
```

- `VK_DEBUG_REPORT_ERROR_BIT_EXT` specifies that the application has violated a valid usage condition of the specification.
- `VK_DEBUG_REPORT_WARNING_BIT_EXT` specifies use of Vulkan that **may** expose an app bug. Such cases may not be immediately harmful, such as a fragment shader outputting to a location with no attachment. Other cases **may** point to behavior that is almost certainly bad when unintended such as using an image whose memory has not been filled. In general if you see a warning but you know that the behavior is intended/desired, then simply ignore the warning.
- `VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT` specifies a potentially non-optimal use of Vulkan, e.g. using `vkCmdClearColorImage` when setting `VkAttachmentDescription::loadOp` to `VK_ATTACHMENT_LOAD_OP_CLEAR` would have worked.
- `VK_DEBUG_REPORT_INFORMATION_BIT_EXT` specifies an informational message such as resource details that may be handy when debugging an application.
- `VK_DEBUG_REPORT_DEBUG_BIT_EXT` specifies diagnostic information from the implementation and layers.

```
typedef VkFlags VkDebugReportFlagsEXT;
```

`VkDebugReportFlagsEXT` is a bitmask type for setting a mask of zero or more `VkDebugReportFlagBitEXT`.

The prototype for the `VkDebugReportCallbackCreateInfoEXT::pfnCallback` function implemented by

the application is:

```
typedef VkBool32 (VKAPI_PTR *PFN_vkDebugReportCallbackEXT)(  
    VkDebugReportFlagsEXT  
    VkDebugReportObjectTypeEXT  
    uint64_t  
    size_t  
    int32_t  
    const char*  
    const char*  
    void*
```

- **flags** specifies the [VkDebugReportFlagBitsEXT](#) that triggered this callback.
- **objectType** is a [VkDebugReportObjectTypeEXT](#) value specifying the type of object being used or created at the time the event was triggered.
- **object** is the object where the issue was detected. If **objectType** is [VK\\_DEBUG\\_REPORT\\_OBJECT\\_TYPE\\_UNKNOWN\\_EXT](#), **object** is undefined.
- **location** is a component (layer, driver, loader) defined value specifying the *location* of the trigger. This is an **optional** value.
- **messageCode** is a layer-defined value indicating what test triggered this callback.
- **pLayerPrefix** is a null-terminated string that is an abbreviation of the name of the component making the callback. **pLayerPrefix** is only valid for the duration of the callback.
- **pMessage** is a null-terminated string detailing the trigger conditions. **pMessage** is only valid for the duration of the callback.
- **pUserData** is the user data given when the [VkDebugReportCallbackEXT](#) was created.

The callback **must** not call [vkDestroyDebugReportCallbackEXT](#).

The callback returns a [VkBool32](#), which is interpreted in a layer-specified manner. The application **should** always return [VK\\_FALSE](#). The [VK\\_TRUE](#) value is reserved for use in layer development.

**object** **must** be a Vulkan object or [VK\\_NULL\\_HANDLE](#). If **objectType** is not [VK\\_DEBUG\\_REPORT\\_OBJECT\\_TYPE\\_UNKNOWN\\_EXT](#) and **object** is not [VK\\_NULL\\_HANDLE](#), **object** **must** be a Vulkan object of the corresponding type associated with **objectType** as defined in [VkDebugReportObjectTypeEXT](#) and [Vulkan Handle Relationship](#).

Possible values passed to the **objectType** parameter of the callback function specified by [VkDebugReportCallbackCreateInfoEXT::pfnCallback](#), specifying the type of object handle being reported, are:

```

typedef enum VkDebugReportObjectTypeEXT {
    VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT = 0,
    VK_DEBUG_REPORT_OBJECT_TYPE_INSTANCE_EXT = 1,
    VK_DEBUG_REPORT_OBJECT_TYPE_PHYSICAL_DEVICE_EXT = 2,
    VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT = 3,
    VK_DEBUG_REPORT_OBJECT_TYPE_QUEUE_EXT = 4,
    VK_DEBUG_REPORT_OBJECT_TYPE_SEMAPHORE_EXT = 5,
    VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_BUFFER_EXT = 6,
    VK_DEBUG_REPORT_OBJECT_TYPE_FENCE_EXT = 7,
    VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_MEMORY_EXT = 8,
    VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT = 9,
    VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_EXT = 10,
    VK_DEBUG_REPORT_OBJECT_TYPE_EVENT_EXT = 11,
    VK_DEBUG_REPORT_OBJECT_TYPE_QUERY_POOL_EXT = 12,
    VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_VIEW_EXT = 13,
    VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_VIEW_EXT = 14,
    VK_DEBUG_REPORT_OBJECT_TYPE_SHADER_MODULE_EXT = 15,
    VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_CACHE_EXT = 16,
    VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_LAYOUT_EXT = 17,
    VK_DEBUG_REPORT_OBJECT_TYPE_RENDER_PASS_EXT = 18,
    VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_EXT = 19,
    VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT_EXT = 20,
    VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_EXT = 21,
    VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_POOL_EXT = 22,
    VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_EXT = 23,
    VK_DEBUG_REPORT_OBJECT_TYPE_FRAMEBUFFER_EXT = 24,
    VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_POOL_EXT = 25,
    VK_DEBUG_REPORT_OBJECT_TYPE_SURFACE_KHR_EXT = 26,
    VK_DEBUG_REPORT_OBJECT_TYPE_SWAPCHAIN_KHR_EXT = 27,
    VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT_EXT = 28,
    VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_KHR_EXT = 29,
    VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_MODE_KHR_EXT = 30,
    VK_DEBUG_REPORT_OBJECT_TYPE_OBJECT_TABLE_NVX_EXT = 31,
    VK_DEBUG_REPORT_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NVX_EXT = 32,
    VK_DEBUG_REPORT_OBJECT_TYPE_VALIDATION_CACHE_EXT_EXT = 33,
    VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_EXT = 1000156000,
    VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_EXT = 1000085000,
    VK_DEBUG_REPORT_OBJECT_TYPE_ACCELERATION_STRUCTURE_NV_EXT = 1000165000,
    VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_EXT =
    VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT_EXT,
    VK_DEBUG_REPORT_OBJECT_TYPE_VALIDATION_CACHE_EXT =
    VK_DEBUG_REPORT_OBJECT_TYPE_VALIDATION_CACHE_EXT_EXT,
    VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR_EXT =
    VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_EXT,
    VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_KHR_EXT =
    VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_EXT,
    VK_DEBUG_REPORT_OBJECT_TYPE_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDebugReportObjectTypeEXT;

```

Table 74. *VkDebugReportObjectTypeEXT* and Vulkan Handle Relationship

<a href="#">VkDebugReportObjectTypeEXT</a>	<a href="#">Vulkan Handle Type</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT</code>	Unknown/Undefined Handle
<code>VK_DEBUG_REPORT_OBJECT_TYPE_INSTANCE_EXT</code>	<a href="#">VkInstance</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_PHYSICAL_DEVICE_EXT</code>	<a href="#">VkPhysicalDevice</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT</code>	<a href="#">VkDevice</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_QUEUE_EXT</code>	<a href="#">VkQueue</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_SEMAPHORE_EXT</code>	<a href="#">VkSemaphore</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_BUFFER_EXT</code>	<a href="#">VkCommandBuffer</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_FENCE_EXT</code>	<a href="#">VkFence</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_MEMORY_EXT</code>	<a href="#">VkDeviceMemory</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT</code>	<a href="#">VkBuffer</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_EXT</code>	<a href="#">VkImage</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_EVENT_EXT</code>	<a href="#">VkEvent</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_QUERY_POOL_EXT</code>	<a href="#">VkQueryPool</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_VIEW_EXT</code>	<a href="#">VkBufferView</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_VIEW_EXT</code>	<a href="#">VkImageView</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_SHADER_MODULE_EXT</code>	<a href="#">VkShaderModule</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_CACHE_EXT</code>	<a href="#">VkPipelineCache</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_LAYOUT_EXT</code>	<a href="#">VkPipelineLayout</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_RENDER_PASS_EXT</code>	<a href="#">VkRenderPass</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_EXT</code>	<a href="#">VkPipeline</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT_EXT</code>	<a href="#">VkDescriptorSetLayout</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_EXT</code>	<a href="#">VkSampler</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_POOL_EXT</code>	<a href="#">VkDescriptorPool</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_EXT</code>	<a href="#">VkDescriptorSet</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_FRAMEBUFFER_EXT</code>	<a href="#">VkFramebuffer</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_POOL_EXT</code>	<a href="#">VkCommandPool</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_SURFACE_KHR_EXT</code>	<a href="#">VkSurfaceKHR</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_SWAPCHAIN_KHR_EXT</code>	<a href="#">VkSwapchainKHR</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT</code>	<a href="#">VkDebugReportCallbackEXT</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_KHR_EXT</code>	<a href="#">VkDisplayKHR</a>
<code>VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_MODE_KHR_EXT</code>	<a href="#">VkDisplayModeKHR</a>

VkDebugReportObjectTypeEXT	Vulkan Handle Type
VK_DEBUG_REPORT_OBJECT_TYPE_OBJECT_TABLE_NVX_EXT	VkObjectTableNVX
VK_DEBUG_REPORT_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NVX_EXT	VkIndirectCommandsLayoutNVX
VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_EXT	VkDescriptorUpdateTemplate

*Note*



The primary expected use of `VK_ERROR_VALIDATION_FAILED_EXT` is for validation layer testing. It is not expected that an application would see this error code during normal use of the validation layers.

To inject its own messages into the debug stream, call:

```
void vkDebugReportMessageEXT(
    VkInstance                                     instance,
    VkDebugReportFlagsEXT                         flags,
    VkDebugReportObjectTypeEXT                    objectType,
    uint64_t                                       object,
    size_t                                         location,
    int32_t                                        messageCode,
    const char*                                    pLayerPrefix,
    const char*                                    pMessage);
```

- `instance` is the debug stream's `VkInstance`.
- `flags` specifies the `VkDebugReportFlagBitsEXT` classification of this event/message.
- `objectType` is a `VkDebugReportObjectTypeEXT` specifying the type of object being used or created at the time the event was triggered.
- `object` this is the object where the issue was detected. `object` can be `VK_NULL_HANDLE` if there is no object associated with the event.
- `location` is an application defined value.
- `messageCode` is an application defined value.
- `pLayerPrefix` is the abbreviation of the component making this event/message.
- `pMessage` is a null-terminated string detailing the trigger conditions.

The call will propagate through the layers and generate callback(s) as indicated by the message's flags. The parameters are passed on to the callback in addition to the `pUserData` value that was defined at the time the callback was registered.

## Valid Usage

- `object` **must** be a Vulkan object or `VK_NULL_HANDLE`
- If `objectType` is not `VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT` and `object` is not `VK_NULL_HANDLE`, `object` **must** be a Vulkan object of the corresponding type associated with `objectType` as defined in `VkDebugReportObjectTypeEXT` and `Vulkan Handle Relationship`.

## Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `flags` **must** be a valid combination of `VkDebugReportFlagBitsEXT` values
- `flags` **must** not be `0`
- `objectType` **must** be a valid `VkDebugReportObjectTypeEXT` value
- `pLayerPrefix` **must** be a null-terminated UTF-8 string
- `pMessage` **must** be a null-terminated UTF-8 string

To destroy a `VkDebugReportCallbackEXT` object, call:

```
void vkDestroyDebugReportCallbackEXT(  
    VkInstance                      instance,  
    VkDebugReportCallbackEXT        callback,  
    const VkAllocationCallbacks* pAllocator);
```

- `instance` the instance where the callback was created.
- `callback` the `VkDebugReportCallbackEXT` object to destroy. `callback` is an externally synchronized object and **must** not be used on more than one thread at a time. This means that `vkDestroyDebugReportCallbackEXT` **must** not be called when a callback is active.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

## Valid Usage

- If `VkAllocationCallbacks` were provided when `callback` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `callback` was created, `pAllocator` **must** be `NULL`

## Valid Usage (Implicit)

- **instance** **must** be a valid `VkInstance` handle
- **callback** **must** be a valid `VkDebugReportCallbackEXT` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **callback** **must** have been created, allocated, or retrieved from `instance`

## Host Synchronization

- Host access to `callback` **must** be externally synchronized

# 39.4. Device Loss Debugging

## 39.4.1. Device Diagnostic Checkpoints

Device execution progress **can** be tracked for the purposes of debugging a device loss by annotating the command stream with application-defined diagnostic checkpoints.

Each diagnostic checkpoint command is executed at two pipeline stages: `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`, and `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`. If the device is lost, the application **can** call `vkGetQueueCheckpointDataNV` to retrieve checkpoint data associated with both pipeline stages, indicating the range of diagnostic checkpoints that are currently in the execution pipeline on the device.

Device diagnostic checkpoints are inserted into the command stream by calling `vkCmdSetCheckpointNV`.

```
void vkCmdSetCheckpointNV(  
    VkCommandBuffer  
    const void*  
                                commandBuffer,  
                                pCheckpointMarker);
```

- `commandBuffer` is the command buffer that will receive the marker
- `pCheckpointMarker` is an opaque application-provided value that will be associated with the checkpoint.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, compute, or transfer operations

## Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types	Pipeline Type
Primary Secondary	Both	Graphics Compute Transfer	

Note that `pCheckpointMarker` is treated as an opaque value. It does not need to be a valid pointer and will not be dereferenced by the implementation.

If the device encounters an error during execution, the implementation will return a `VK_ERROR_DEVICE_LOST` error to the application at a certain point during host execution. When this happens, the application **can** call `vkGetQueueCheckpointDataNV` to retrieve information on the most recent diagnostic checkpoints that were executed by the device.

```
void vkGetQueueCheckpointDataNV(  
    VkQueue queue,  
    uint32_t* pCheckpointDataCount,  
    VkCheckpointDataNV* pCheckpointData);
```

- `queue` is the `VkQueue` object the caller would like to retrieve checkpoint data for
- `pCheckpointDataCount` is a pointer to an integer related to the number of checkpoint markers available or queried, as described below.
- `pCheckpointData` is either `NULL` or a pointer to an array of `VkCheckpointDataNV` structures.

If `pCheckpointData` is `NULL`, then the number of checkpoint markers available is returned in `pCheckpointDataCount`.

Otherwise, `pCheckpointDataCount` **must** point to a variable set by the user to the number of elements in the `pCheckpointData` array, and on return the variable is overwritten with the number of

structures actually written to `pCheckpointData`.

If `pCheckpointDataCount` is less than the number of checkpoint markers available, at most `pCheckpointDataCount` structures will be written.

## Valid Usage

- The device that `queue` belongs to **must** be in the lost state

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- `pCheckpointDataCount` **must** be a valid pointer to a `uint32_t` value
- If the value referenced by `pCheckpointDataCount` is not `0`, and `pCheckpointData` is not `NULL`, `pCheckpointData` **must** be a valid pointer to an array of `pCheckpointDataCount` `VkCheckpointDataNV` structures

The `VkCheckpointDataNV` structure is defined as:

```
typedef struct VkCheckpointDataNV {
    VkStructureType          sType;
    void*                    pNext;
    VkPipelineStageFlagBits   stage;
    void*                    pCheckpointMarker;
} VkCheckpointDataNV;
```

- `sType` is the type of this structure
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `stage` indicates which pipeline stage the checkpoint marker data refers to.
- `pCheckpointMarker` contains the value of the last checkpoint marker executed in the stage that `stage` refers to.

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_CHECKPOINT_DATA_NV`
- `pNext` **must** be `NULL`

Note that the stages at which a checkpoint marker **can** be executed are implementation-defined and **can** be queried by calling `vkGetPhysicalDeviceQueueFamilyProperties2`.

# Appendix A: Vulkan Environment for SPIR-V

Shaders for Vulkan are defined by the [Khronos SPIR-V Specification](#) as well as the [Khronos SPIR-V Extended Instructions for GLSL Specification](#). This appendix defines additional SPIR-V requirements applying to Vulkan shaders.

## Versions and Formats

A Vulkan 1.1 implementation **must** support the 1.0, 1.1, 1.2, and 1.3 versions of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for GLSL. If the `VK_KHR_spirv_1_4` extension is enabled, the implementation **must** additionally support the 1.4 version of SPIR-V.

A SPIR-V module passed into `vkCreateShaderModule` is interpreted as a series of 32-bit words in host endianness, with literal strings packed as described in section 2.2 of the SPIR-V Specification. The first few words of the SPIR-V module **must** be a magic number and a SPIR-V version number, as described in section 2.3 of the SPIR-V Specification.

## Capabilities

The SPIR-V capabilities listed below **must** be supported if the corresponding feature or extension is enabled, or if no features or extensions are listed for that capability. Extensions are only listed when there is not also a feature bit associated with that capability.

*Table 75. List of SPIR-V Capabilities and enabling features or extensions*

SPIR-V OpCapability	Vulkan feature or extension name
Matrix	
Shader	
InputAttachment	
Sampled1D	
Image1D	
SampledBuffer	
ImageBuffer	
ImageQuery	
DerivativeControl	
Geometry	<code>geometryShader</code>
Tessellation	<code>tessellationShader</code>
Float64	<code>shaderFloat64</code>
Int64	<code>shaderInt64</code>
Int64Atomics	<code>VK_KHR_shader_atomic_int64</code>
Int16	<code>shaderInt16</code>
TessellationPointSize	<code>shaderTessellationAndGeometryPointSize</code>
GeometryPointSize	<code>shaderTessellationAndGeometryPointSize</code>
ImageGatherExtended	<code>shaderImageGatherExtended</code>

SPIR-V OpCapability	Vulkan feature or extension name
StorageImageMultisample	shaderStorageImageMultisample
UniformBufferArrayDynamicIndexing	shaderUniformBufferArrayDynamicIndexing
SampledImageArrayDynamicIndexing	shaderSampledImageArrayDynamicIndexing
StorageBufferArrayDynamicIndexing	shaderStorageBufferArrayDynamicIndexing
StorageImageArrayDynamicIndexing	shaderStorageImageArrayDynamicIndexing
ClipDistance	shaderClipDistance
CullDistance	shaderCullDistance
ImageCubeArray	imageCubeArray
SampleRateShading	sampleRateShading
SparseResidency	shaderResourceResidency
MinLod	shaderResourceMinLod
SampledCubeArray	imageCubeArray
ImageMSArray	shaderStorageImageMultisample
StorageImageExtendedFormats	
InterpolationFunction	sampleRateShading
StorageImageReadWithoutFormat	shaderStorageImageReadWithoutFormat
StorageImageWriteWithoutFormat	shaderStorageImageWriteWithoutFormat
MultiViewport	multiViewport
DrawParameters	shaderDrawParameters or VK_KHR_shader_draw_parameters
MultiView	
DeviceGroup	
VariablePointersStorageBuffer	variablePointersStorageBuffer
VariablePointers	variablePointers
ShaderClockKHR	VK_KHR_shader_clock
StencilExportEXT	VK_EXT_shader_stencil_export
SubgroupBallotKHR	VK_EXT_shader_subgroup_ballot
SubgroupVoteKHR	VK_EXT_shader_subgroup_vote
ImageReadWriteLodAMD	VK_AMD_shader_image_load_store_lod
ImageGatherBiasLodAMD	VK_AMD_texture_gather_bias_lod
FragmentMaskAMD	VK_AMD_shader_fragment_mask
SampleMaskOverrideCoverageNV	VK_NV_sample_mask_override_coverage
GeometryShaderPassthroughNV	VK_NV_geometry_shader_passthrough
ShaderViewportIndexLayerEXT	VK_EXT_shader_viewport_index_layer
ShaderViewportIndexLayerNV	VK_NV_viewport_array2
ShaderViewportMaskNV	VK_NV_viewport_array2
PerViewAttributesNV	VK_NVX_multiview_per_view_attributes

SPIR-V OpCapability	Vulkan feature or extension name
StorageBuffer16BitAccess	StorageBuffer16BitAccess
UniformAndStorageBuffer16BitAccess	UniformAndStorageBuffer16BitAccess
StoragePushConstant16	storagePushConstant16
StorageInputOutput16	storageInputOutput16
GroupNonUniform	VK_SUBGROUP_FEATURE_BASIC_BIT
GroupNonUniformVote	VK_SUBGROUP_FEATURE_VOTE_BIT
GroupNonUniformArithmetic	VK_SUBGROUP_FEATURE_ARITHMETIC_BIT
GroupNonUniformBallot	VK_SUBGROUP_FEATURE_BALLOT_BIT
GroupNonUniformShuffle	VK_SUBGROUP_FEATURE_SHUFFLE_BIT
GroupNonUniformShuffleRelative	VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT
GroupNonUniformClustered	VK_SUBGROUP_FEATURE_CLUSTERED_BIT
GroupNonUniformQuad	VK_SUBGROUP_FEATURE_QUAD_BIT
GroupNonUniformPartitionedNV	VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV
SampleMaskPostDepthCoverage	VK_EXT_post_depth_coverage
ShaderNonUniformEXT	VK_EXT_descriptor_indexing
RuntimeDescriptorArrayEXT	runtimeDescriptorArray
InputAttachmentArrayDynamicIndexingEXT	shaderInputAttachmentArrayDynamicIndexing
UniformTexelBufferArrayDynamicIndexingEXT	shaderUniformTexelBufferArrayDynamicIndexing
StorageTexelBufferArrayDynamicIndexingEXT	shaderStorageTexelBufferArrayDynamicIndexing
UniformBufferArrayNonUniformIndexingEXT	shaderUniformBufferArrayNonUniformIndexing
SampledImageArrayNonUniformIndexingEXT	shaderSampledImageArrayNonUniformIndexing
StorageBufferArrayNonUniformIndexingEXT	shaderStorageBufferArrayNonUniformIndexing
StorageImageArrayNonUniformIndexingEXT	shaderStorageImageArrayNonUniformIndexing
InputAttachmentArrayNonUniformIndexingEXT	shaderInputAttachmentArrayNonUniformIndexing
UniformTexelBufferArrayNonUniformIndexingEXT	shaderUniformTexelBufferArrayNonUniformIndexing
StorageTexelBufferArrayNonUniformIndexingEXT	shaderStorageTexelBufferArrayNonUniformIndexing
Float16	shaderFloat16 or VK_AMD_gpu_shader_half_float
Int8	shaderInt8
StorageBuffer8BitAccess	StorageBuffer8BitAccess
UniformAndStorageBuffer8BitAccess	UniformAndStorageBuffer8BitAccess

SPIR-V OpCapability	Vulkan feature or extension name
StoragePushConstant8	StoragePushConstant8
VulkanMemoryModelKHR	vulkanMemoryModel
VulkanMemoryModelDeviceScopeKHR	vulkanMemoryModelDeviceScope
DenormPreserve	shaderDenormPreserveFloat16, shaderDenormPreserveFloat32, shaderDenormPreserveFloat64
DenormFlushToZero	shaderDenormFlushToZeroFloat16, shaderDenormFlushToZeroFloat32, shaderDenormFlushToZeroFloat64
SignedZeroInfNanPreserve	shaderSignedZeroInfNanPreserveFloat16, shaderSignedZeroInfNanPreserveFloat32, shaderSignedZeroInfNanPreserveFloat64
RoundingModeRTE	shaderRoundingModeRTEFloat16, shaderRoundingModeRTEFloat32, shaderRoundingModeRTEFloat64
RoundingModeRTZ	shaderRoundingModeRTZFloat16, shaderRoundingModeRTZFloat32, shaderRoundingModeRTZFloat64
ComputeDerivativeGroupQuadsNV	computeDerivativeGroupQuads
ComputeDerivativeGroupLinearNV	computeDerivativeGroupLinear
FragmentBarycentricNV	fragmentShaderBarycentric
ImageFootprintNV	imageFootprint
ShadingRateImageNV	shadingRateImage
MeshShadingNV	VK_NV_mesh_shader
RayTracingNV	VK_NV_ray_tracing
TransformFeedback	transformFeedback
GeometryStreams	geometryStreams
FragmentDensityEXT	fragmentDensityMap
PhysicalStorageBufferAddresses	bufferDeviceAddress , VkPhysicalDeviceBufferDeviceAddressFeaturesEXT ::bufferDeviceAddress
CooperativeMatrixNV	cooperativeMatrix
ShaderIntegerFunctions2INTEL	shaderIntegerFunctions2
ShaderSMBuiltinsNV	shaderSMBuiltins
FragmentShaderSampleInterlockEXT	fragmentShaderSampleInterlock
FragmentShaderPixelInterlockEXT	fragmentShaderPixelInterlock
FragmentShaderShadingRateInterlockEXT	fragmentShaderShadingRateInterlock, shadingRateImage
DemoteToHelperInvocationEXT	shaderDemoteToHelperInvocation

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the

[SPV\\_KHR\\_variable\\_pointers](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_shader\\_explicit\\_vertex\\_parameter](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_gcn\\_shader](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_gpu\\_shader\\_half\\_float](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_gpu\\_shader\\_int16](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_shader\\_ballot](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_shader\\_fragment\\_mask](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_shader\\_image\\_load\\_store\\_lod](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_shader\\_trinary\\_minmax](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_AMD\\_texture\\_gather\\_bias\\_lod](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_KHR\\_shader\\_draw\\_parameters](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_KHR\\_8bit\\_storage](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_KHR\\_16bit\\_storage](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_KHR\\_shader\\_clock](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_KHR\\_float\\_controls](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_KHR\\_storage\\_buffer\\_storage\\_class](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_KHR\\_post\\_depth\\_coverage](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the [SPV\\_EXT\\_shader\\_stencil\\_export](#) SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_KHR_shader_ballot` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_KHR_subgroup_vote` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_sample_mask_override_coverage` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_geometry_shader_passthrough` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_mesh_shader` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_viewport_array2` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_shader_subgroup_partitioned` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_EXT_shader_viewport_index_layer` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NVX_multiview_per_view_attributes` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_EXT_descriptor_indexing` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_KHR_vulkan_memory_model` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_compute_shader_derivatives` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_fragment_shader_barycentric` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_shader_image_footprint` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_shading_rate` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_ray_tracing` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_GOOGLE_hlsl_functionality1` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the

`SPV_GOOGLE_user_type` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_GOOGLE_decorate_string` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_EXT_fragment_invocation_density` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_KHR_physical_storage_buffer` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_EXT_physical_storage_buffer` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_cooperative_matrix` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_NV_shader_sm_builtins` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_EXT_fragment_shader_interlock` SPIR-V extension.

The application **can** pass a SPIR-V module to `vkCreateShaderModule` that uses the `SPV_EXT_demote_to_helper_invocation` SPIR-V extension.

The application **must** not pass a SPIR-V module containing any of the following to `vkCreateShaderModule`:

- any OpCapability not listed above,
- an unsupported capability, or
- a capability which corresponds to a Vulkan feature or extension which has not been enabled.

## Validation Rules within a Module

A SPIR-V module passed to `vkCreateShaderModule` **must** conform to the following rules:

- Every entry point **must** have no return value and accept no arguments.
- Recursion: The static function-call graph for an entry point **must** not contain cycles.
- The **Logical** addressing model **must** be selected.
- **Scope** for execution **must** be limited to:
  - **Workgroup**
  - **Subgroup**
- **Scope** for memory **must** be limited to:
  - **Device**
    - If `vulkanMemoryModel` is enabled and `vulkanMemoryModelDeviceScope` is not enabled, **Device**

scope **must** not be used.

- If `vulkanMemoryModel` is not enabled, **Device** scope only extends to the queue family, not the whole device.

- **QueueFamilyKHR**

- If `vulkanMemoryModel` is not enabled, **QueueFamilyKHR** **must** not be used.

- **Workgroup**

- **Subgroup**

- **Invocation**

- **Scope for Non Uniform Group Operations** **must** be limited to:

- **Subgroup**

- **Storage Class** **must** be limited to:

- **UniformConstant**

- **Input**

- **Uniform**

- **Output**

- **Workgroup**

- **Private**

- **Function**

- **PushConstant**

- **Image**

- **StorageBuffer**

- **RayPayloadNV**

- **IncomingRayPayloadNV**

- **HitAttributeNV**

- **CallableDataNV**

- **IncomingCallableDataNV**

- **ShaderRecordBufferNV**

- **PhysicalStorageBuffer**

- Memory semantics **must** obey the following rules:

- **Acquire** **must** not be used with `OpAtomicStore`.

- **Release** **must** not be used with `OpAtomicLoad`.

- **AcquireRelease** **must** not be used with `OpAtomicStore` or `OpAtomicLoad`.

- Sequentially consistent atomics and barriers are not supported and **SequentiallyConsistent** is treated as **AcquireRelease**. **SequentiallyConsistent** **should** not be used.

- `OpMemoryBarrier` **must** use one of **Acquire**, **Release**, **AcquireRelease**, or

**SequentiallyConsistent** and **must** include at least one storage class.

- If the semantics for **OpControlBarrier** includes one of **Acquire**, **Release**, **AcquireRelease**, or **SequentiallyConsistent**, then it **must** include at least one storage class.
- **SubgroupMemory**, **CrossWorkgroupMemory**, and **AtomicCounterMemory** are ignored.
- Any **OpVariable** with an **Initializer** operand **must** have one of the following as its **Storage Class** operand:
  - **Output**
  - **Private**
  - **Function**
- Scope for **OpReadClockKHR** must be limited to:
  - **Subgroup**
    - if **shaderSubgroupClock** is not enabled, the **Subgroup** scope **must** not be used.
  - **Device**
    - if **shaderDeviceClock** is not enabled, the **Device** scope **must** not be used.
- The **OriginLowerLeft** execution mode **must** not be used; fragment entry points **must** declare **OriginUpperLeft**.
- The **PixelCenterInteger** execution mode **must** not be used. Pixels are always centered at half-integer coordinates.
- Any variable in the **UniformConstant** storage class **must** be typed as either:
  - **OpTypeImage**
  - **OpTypeSampler**
  - **OpTypeSampledImage**
  - An array of one of these types.
- Images and Samplers
  - **OpTypeImage** **must** declare a scalar 32-bit float or 32-bit integer type for the “Sampled Type”. (**RelaxedPrecision** **can** be applied to a sampling instruction and to the variable holding the result of a sampling instruction.)
  - If the **Sampled Type** of an **OpTypeImage** declaration does not match the numeric format of the corresponding resource in type, as shown in the *SPIR-V Sampled Type* column of the **Interpretation of Numeric Format** table, the values obtained by reading or sampling from the image are undefined.
  - If the signedness of any read or sample operation does not match the signedness of the corresponding resource then the values obtained are undefined.
  - **OpTypeImage** **must** have a “Sampled” operand of 1 (sampled image) or 2 (storage image).
  - If **shaderStorageImageReadWithoutFormat** is not enabled and an **OpTypeImage** has “Image Format” operand of **Unknown**, any variables created with the given type must be decorated with **NonReadable**.
  - If **shaderStorageImageWriteWithoutFormat** is not enabled and an **OpTypeImage** has “Image Format” operand of **Unknown**, any variables created with the given type must be decorated

with `NonWritable`.

- `OpImageQuerySizeLod`, and `OpImageQueryLevels` **must** only consume an “Image” operand whose type has its “Sampled” operand set to 1.
- The (u,v) coordinates used for a `SubpassData` **must** be the <id> of a constant vector (0,0), or if a layer coordinate is used, **must** be a vector that was formed with constant 0 for the u and v components.
- The “Depth” operand of `OpTypeImage` is ignored.
- Objects of types `OpTypeImage`, `OpTypeSampler`, `OpTypeSampledImage`, and arrays of these types **must** not be stored to or modified.
- Any image operation **must** use at most one of the `Offset`, `ConstOffset`, and `ConstOffsets` image operands.
- Image operand `Offset` **must** only be used with `OpImage*Gather` instructions.
- The “Component” operand of `OpImageGather`, and `OpImageSparseGather` **must** be the <id> of a constant instruction.
- Structure types **must** not contain opaque types.
- Decorations
  - Any `BuiltIn` decoration not listed in [Built-In Variables](#) **must** not be used.
  - Any `BuiltIn` decoration that corresponds only to Vulkan features or extensions that have not been enabled **must** not be used.
  - The `GLSLShared` and `GLSLPacked` decorations **must** not be used.
  - The `Flat`, `NoPerspective`, `Sample`, and `Centroid` decorations **must** not be used on variables with storage class other than `Input` or on variables used in the interface of non-fragment shader entry points.
  - The `Patch` decoration **must** not be used on variables in the interface of a vertex, geometry, or fragment shader stage’s entry point.
  - The `ViewportRelativeNV` decoration **must** only be used on a variable decorated with `Layer` in the vertex, tessellation evaluation, or geometry shader stages.
  - The `ViewportRelativeNV` decoration **must** not be used unless a variable decorated with one of `ViewportIndex` or `ViewportMaskNV` is also statically used by the same `OpEntryPoint`.
  - The `ViewportMaskNV` and `ViewportIndex` decorations **must** not both be statically used by one or more `OpEntryPoint`’s that form the vertex processing stages of a graphics pipeline.
  - Only the round-to-nearest-even and the round-towards-zero rounding modes **can** be used for the `FPRoundingMode` decoration.
  - The `FPRoundingMode` decoration **can** only be used for the floating-point conversion instructions as described in the `SPV_KHR_16bit_storage` SPIR-V extension.
  - `DescriptorSet` and `Binding` decorations **must** obey the constraints on storage class, type, and descriptor type described in [DescriptorSet and Binding Assignment](#)
  - Variables decorated with `Invariant` and variables with structure types that have any members decorated with `Invariant` **must** be in the `Output` or `Input` storage class. `Invariant` used on an `Input` storage class variable or structure member has no effect.

- **OpTypeRuntimeArray** **must** only be used for:
  - the last member of an **OpTypeStruct** that is in the **StorageBuffer** storage class decorated as **Block**, or that is in the **PhysicalStorageBuffer** storage class decorated as **Block**, or that is in the **Uniform** storage class decorated as **BufferBlock**.
  - If the **RuntimeDescriptorArrayEXT** capability is supported, an array of variables with storage class **Uniform**, **StorageBuffer**, or **UniformConstant**, or for the outermost dimension of an array of arrays of such variables.
- Linkage: See [Shader Interfaces](#) for additional linking and validation rules.
- If **OpControlBarrier** is used in fragment, vertex, tessellation evaluation, or geometry stages, the execution Scope **must** be **Subgroup**.
- Compute Shaders
  - For each compute shader entry point, either a **LocalSize** execution mode or an object decorated with the **WorkgroupSize** decoration **must** be specified.
  - For compute shaders using the **DerivativeGroupQuadsNV** execution mode, the first two dimensions of the local workgroup size **must** be a multiple of two.
  - For compute shaders using the **DerivativeGroupLinearNV** execution mode, the product of the dimensions of the local workgroup size **must** be a multiple of four.
- “Result Type” for **Non Uniform Group Operations** **must** be limited to 32-bit floating-point, 32-bit integer, boolean, or vectors of these types.
  - If the **Float64** capability is enabled, 64-bit floating-point and vector of 64-bit floating-point types are also permitted.
  - If the **Int8** capability is enabled and the **shaderSubgroupExtendedTypes** feature is **VK\_TRUE**, 8-bit integer and vector of 8-bit integer types are also permitted.
  - If the **Int16** capability is enabled and the **shaderSubgroupExtendedTypes** feature is **VK\_TRUE**, 16-bit integer and vector of 16-bit integer types are also permitted.
  - If the **Int64** capability is enabled and the **shaderSubgroupExtendedTypes** feature is **VK\_TRUE**, 64-bit integer and vector of 64-bit integer types are also permitted.
  - If the **Float16** capability is enabled and the **shaderSubgroupExtendedTypes** feature is **VK\_TRUE**, 16-bit floating-point and vector of 16-bit floating-point types are also permitted.
- The “Id” operand of **OpGroupNonUniformBroadcast** **must** be the <id> of a constant instruction.
- If **OpGroupNonUniformBallotBitCount** is used, the group operation **must** be one of:
  - **Reduce**
  - **InclusiveScan**
  - **ExclusiveScan**
- Atomic instructions **must** declare a scalar 32-bit integer type, or a scalar 64-bit integer type if the **Int64Atomics** capability is enabled, for the value pointed to by *Pointer*.
  - **shaderBufferInt64Atomics** **must** be enabled for 64-bit integer atomic operations to be supported on a *Pointer* with a **Storage Class** of **StorageBuffer** or **Uniform**.
  - **shaderSharedInt64Atomics** **must** be enabled for 64-bit integer atomic operations to be

supported on a *Pointer* with a **Storage Class** of **Workgroup**.

- The *Pointer* operand of all atomic instructions **must** have a **Storage Class** limited to:
  - **Uniform**
  - **Workgroup**
  - **Image**
  - **StorageBuffer**
- If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not dynamically uniform, then the operand corresponding to that resource (e.g. the pointer or sampled image operand) **must** be decorated with `NonUniformEXT`.
- If `denormBehaviorIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY_KHR`, then the entry point **must** use the same denormals execution mode for both 16-bit and 64-bit floating-point types.
- If `denormBehaviorIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE_KHR`, then the entry point **must** use the same denormals execution mode for all floating-point types.
- If `roundingModeIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY_KHR`, then the entry point **must** use the same rounding execution mode for both 16-bit and 64-bit floating-point types.
- If `roundingModeIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE_KHR`, then the entry point **must** use the same rounding execution mode for all floating-point types.
- If `shaderSignedZeroInfNanPreserveFloat16` is `VK_FALSE`, then `SignedZeroInfNanPreserve` for 16-bit floating-point type **must** not be used.
- If `shaderSignedZeroInfNanPreserveFloat32` is `VK_FALSE`, then `SignedZeroInfNanPreserve` for 32-bit floating-point type **must** not be used.
- If `shaderSignedZeroInfNanPreserveFloat64` is `VK_FALSE`, then `SignedZeroInfNanPreserve` for 64-bit floating-point type **must** not be used.
- If `shaderDenormPreserveFloat16` is `VK_FALSE`, then `DenormPreserve` for 16-bit floating-point type **must** not be used.
- If `shaderDenormPreserveFloat32` is `VK_FALSE`, then `DenormPreserve` for 32-bit floating-point type **must** not be used.
- If `shaderDenormPreserveFloat64` is `VK_FALSE`, then `DenormPreserve` for 64-bit floating-point type **must** not be used.
- If `shaderDenormFlushToZeroFloat16` is `VK_FALSE`, then `DenormFlushToZero` for 16-bit floating-point type **must** not be used.
- If `shaderDenormFlushToZeroFloat32` is `VK_FALSE`, then `DenormFlushToZero` for 32-bit floating-point type **must** not be used.
- If `shaderDenormFlushToZeroFloat64` is `VK_FALSE`, then `DenormFlushToZero` for 64-bit floating-point type **must** not be used.
- If `shaderRoundingModeRTEFloat16` is `VK_FALSE`, then `RoundingModeRTE` for 16-bit floating-point type **must** not be used.

- If `shaderRoundingModeRTEFloat32` is `VK_FALSE`, then `RoundingModeRTE` for 32-bit floating-point type **must** not be used.
- If `shaderRoundingModeRTEFloat64` is `VK_FALSE`, then `RoundingModeRTE` for 64-bit floating-point type **must** not be used.
- If `shaderRoundingModeRTZFloat16` is `VK_FALSE`, then `RoundingModeRTZ` for 16-bit floating-point type **must** not be used.
- If `shaderRoundingModeRTZFloat32` is `VK_FALSE`, then `RoundingModeRTZ` for 32-bit floating-point type **must** not be used.
- If `shaderRoundingModeRTZFloat64` is `VK_FALSE`, then `RoundingModeRTZ` for 64-bit floating-point type **must** not be used.
- The `Offset` plus size of the type of each variable, in the output interface of the entry point being compiled, decorated with `XfbBuffer` **must** not be greater than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferDataSize`
- For any given `XfbBuffer` value, define the buffer data size to be smallest number of bytes such that, for all outputs decorated with the same `XfbBuffer` value, the size of the output interface variable plus the `Offset` is less than or equal to the buffer data size. For a given `Stream`, the sum of all the buffer data sizes for all buffers writing to that stream the **must** not exceed `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreamDataSize`
- Output variables or block members decorated with `Offset` that have a 64-bit type, or a composite type containing a 64-bit type, **must** specify an `Offset` value aligned to a 8 byte boundary
- Any output block or block member decorated with `Offset` containing a 64-bit type consumes a multiple of 8 bytes
- The size of any output block containing any member decorated with `Offset` that is a 64-bit type **must** be a multiple of 8
- The first member of an output block that specifies a `Offset` decoration **must** specify a `Offset` value that is aligned to an 8 byte boundary if that block contains any member decorated with `Offset` and is a 64-bit type
- Output variables or block members decorated with `Offset` that have a 32-bit type, or a composite type contains a 32-bit type, **must** specify an `Offset` value aligned to a 4 byte boundary
- Output variables, blocks or block members decorated with `Offset` **must** only contain base types that have components that are either 32-bit or 64-bit in size
- The `Stream` value to `OpEmitStreamVertex` and `OpEndStreamPrimitive` **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams`
- If the geometry shader emits to more than one vertex stream and `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackStreamsLinesTriangles` is `VK_FALSE`, then execution mode **must** be `OutputPoints`
- Only variables or block members in the output interface decorated with `Offset` **can** be captured for transform feedback, and those variables or block memebers **must** also be decorated with `XfbBuffer` and `XfbStride`, or inherit `XfbBuffer` and `XfbStride` decorations from a block containing them

- All variables or block members in the output interface of the entry point being compiled decorated with a specific `XfbBuffer` value **must** all be decorated with identical `XfbStride` values
- If any variables or block members in the output interface of the entry point being compiled are decorated with `Stream`, then all variables belonging to the same `XfbBuffer` must specify the same `Stream` value
- Output variables, blocks or block members that are not decorated with `Stream` default to vertex stream zero
- For any two variables or block members in the output interface of the entry point being compiled with the same `XfbBuffer` value, the ranges determined by the `Offset` decoration and the size of the type **must** not overlap
- The stream number value to `Stream` **must** be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams`
- The XFB Stride value to `XfbStride` **must** be less than or equal to `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferDataStride`
- `RayPayloadNV` storage class **must** only be used in ray generation, any-hit, closest hit or miss shaders.
- `IncomingRayPayloadNV` storage class **must** only be used in closest hit, any-hit, or miss shaders.
- `HitAttributeNV` storage class **must** only be used in intersection, any-hit, or closest hit shaders.
- `CallableDataNV` storage class **must** only be used in ray generation, closest hit, miss, and callable shaders.
- `IncomingCallableDataNV` storage class must only be used in callable shaders.
- The `Base` operand of `OpPtrAccessChain` **must** point to one of the following storage classes:
  - `Workgroup`, if `VariablePointers` is enabled.
  - `StorageBuffer`, if `VariablePointers` or `VariablePointersStorageBuffer` is enabled.
  - `PhysicalStorageBuffer`, if the `PhysicalStorageBuffer64` addressing model is enabled.
- If the `PhysicalStorageBuffer64` addressing model is enabled:
  - Any load or store through a physical pointer type **must** be aligned to a multiple of the size of the largest scalar type in the pointed-to type.
  - All instructions that support memory access operands and that use a physical pointer **must** include the `Aligned` operand.
  - The pointer value of a memory access instruction must be at least as aligned as specified by the `Aligned` memory access operand.
  - Any access chain instruction that accesses into a `RowMajor` matrix **must** only be used as the `Pointer` operand to `OpLoad` or `OpStore`.
  - `OpConvertUToPtr` and `OpConvertPtrToU` **must** use an integer type whose `Width` is 64.
- For `OpCooperativeMatrixLoadNV` and `OpCooperativeMatrixStoreNV` instructions, the `Pointer` and `Stride` operands **must** be aligned to at least the lesser of 16 bytes or the natural alignment of a row or column (depending on `ColumnMajor`) of the matrix (where the natural alignment is the number of columns/rows multiplied by the component size).

- For `OpTypeCooperativeMatrixNV`, the component type, scope, number of rows, and number of columns **must** match one of the matrices in any of the supported `VkCooperativeMatrixPropertiesNV`.
- For `OpCooperativeMatrixMulAddNV`, the `Result`, `A`, `B`, and `C` matrices **must** all have types that satisfy the same supported `VkCooperativeMatrixPropertiesNV`. That is, for one supported supported `VkCooperativeMatrixPropertiesNV`, all of the following **must** hold:
  - The type of `A` must have `MSize` rows and `KSize` columns and have a component type that matches `AType`.
  - The type of `B` must have `KSize` rows and `NSize` columns and have a component type that matches `BType`.
  - The type of `C` must have `MSize` rows and `NSize` columns and have a component type that matches `CType`.
  - The type of `Result` must have `MSize` rows and `NSize` columns and have a component type that matches `DType`.
  - The type of `A`, `B`, `C`, and `Result` must all have a scope of `scope`.
- `OpTypeCooperativeMatrixNV` and `OpCooperativeMatrix*` instructions **must** not be used in shader stages not included in `VkPhysicalDeviceCooperativeMatrixPropertiesNV::cooperativeMatrixSupportedStages`.

## Precision and Operation of SPIR-V Instructions

The following rules apply to half, single, and double-precision floating point instructions:

- Positive and negative infinities and positive and negative zeros are generated as dictated by [IEEE 754](#), but subject to the precisions allowed in the following table.
- Dividing a non-zero by a zero results in the appropriately signed [IEEE 754](#) infinity.
- Signaling NaNs are not required to be generated and exceptions are never raised. Signaling NaN **may** be converted to quiet NaNs values by any floating point instruction.
- The following instructions **must** not flush denormalized values: `OpConstant`, `OpConstantComposite`, `OpSpecConstant`, `OpSpecConstantComposite`, `OpLoad`, `OpStore`, `OpBitcast`, `OpPhi`, `OpSelect`, `OpFunctionCall`, `OpReturnValue`, `OpVectorExtractDynamic`, `OpVectorInsertDynamic`, `OpVectorShuffle`, `OpCompositeConstruct`, `OpCompositeExtract`, `OpCompositeInsert`, `OpCopyMemory`, `OpCopyObject`.
- By default, the implementation **may** perform optimizations on half, single, or double-precision floating-point instructions respectively that ignore sign of a zero, or assume that arguments and results are not Nans or  $\pm\infty$ , this does not apply to `OpIsNaN` and `OpIsInf`, which **must** always correctly detect Nans and  $\pm\infty$ . If the entry point is declared with the `SignedZeroInfNanPreserve` execution mode, then sign of a zero, Nans, and  $\pm\infty$  **must** not be ignored.
  - The following core SPIR-V instructions **must** respect the `SignedZeroInfNanPreserve` execution mode: `OpPhi`, `OpSelect`, `OpReturnValue`, `OpVectorExtractDynamic`, `OpVectorInsertDynamic`, `OpVectorShuffle`, `OpCompositeConstruct`, `OpCompositeExtract`, `OpCompositeInsert`, `OpCopyObject`, `OpTranspose`, `OpFConvert`, `OpFNegate`, `OpFAdd`, `OpFSub`, `OpFMul`, `OpStore`. This execution mode **must** also be respected by `OpLoad` except for loads from the `Input` storage class in the fragment

shader stage with the floating-point result type. Other SPIR-V instructions **may** also respect the `SignedZeroInfNanPreserve` execution mode.

- Denormalized values are supported.
  - By default, any half, single, or double-precision denormalized value input into a shader or potentially generated by any instruction (except those listed above) or any extended instructions for GLSL in a shader **may** be flushed to zero.
  - If the entry point is declared with the `DenormFlushToZero` execution mode then for the affected instructions the denormalized result **must** be flushed to zero and the denormalized operands **may** be flushed to zero. Denormalized values obtained via unpacking an integer into a vector of values with smaller bit width and interpreting those values as floating-point numbers **must** be flushed to zero.
  - The following core SPIR-V instructions **must** respect the `DenormFlushToZero` execution mode: `OpSpecConstantOp` (with opcode `OpFConvert`), `OpFConvert`, `OpFNegate`, `OpFAdd`, `OpFSub`, `OpFMul`, `OpFDiv`, `OpFRem`, `OpFMod`, `OpVectorTimesScalar`, `OpMatrixTimesScalar`, `OpVectorTimesMatrix`, `OpMatrixTimesVector`, `OpMatrixTimesMatrix`, `OpOuterProduct`, `OpDot`; and the following extended instructions for GLSL: `Round`, `RoundEven`, `Trunc`, `FAbs`, `Floor`, `Ceil`, `Fract`, `Radians`, `Degrees`, `Sin`, `Cos`, `Tan`, `Asin`, `Acos`, `Atan`, `Sinh`, `Cosh`, `Tanh`, `Asinh`, `Acosh`, `Atanh`, `Atan2`, `Pow`, `Exp`, `Log`, `Exp2`, `Log2`, `Sqrt`, `InverseSqrt`, `Determinant`, `MatrixInverse`, `Modf`, `ModfStruct`, `FMin`, `FMax`, `FClamp`, `FMix`, `Step`, `SmoothStep`, `Fma`, `UnpackHalf2x16`, `UnpackDouble2x32`, `Length`, `Distance`, `Cross`, `Normalize`, `FaceForward`, `Reflect`, `Refract`, `NMin`, `NMax`, `NClamp`. Other SPIR-V instructions (except those excluded above) **may** also flush denormalized values.
  - The following core SPIR-V instructions **must** respect the `DenormPreserve` execution mode: `OpTranspose`, `OpSpecConstantOp`, `OpFConvert`, `OpFNegate`, `OpFAdd`, `OpFSub`, `OpFMul`, `OpVectorTimesScalar`, `OpMatrixTimesScalar`, `OpVectorTimesMatrix`, `OpMatrixTimesVector`, `OpMatrixTimesMatrix`, `OpOuterProduct`, `OpDot`, `OpFOrdEqual`, `OpFUnordEqual`, `OpFOrdNotEqual`, `OpFUnordNotEqual`, `OpFOrdLessThan`, `OpFUnordLessThan`, `OpFOrdGreaterThanOrEqual`, `OpFUnordGreaterThanOrEqual`, `OpFOrdLessThanEqual`, `OpFUnordLessThanEqual`, `OpFOrdGreaterThanOrEqual`, `OpFUnordGreaterThanOrEqual`; and the following extended instructions for GLSL: `FAbs`, `FSign`, `Radians`, `Degrees`, `FMin`, `FMax`, `FClamp`, `FMix`, `Fma`, `PackHalf2x16`, `PackDouble2x32`, `UnpackHalf2x16`, `UnpackDouble2x32`, `NMin`, `NMax`, `NClamp`. Other SPIR-V instructions **may** also preserve denorm values.

The precision of double-precision instructions is at least that of single precision.

The precision of operations is defined either in terms of rounding, as an error bound in ULP, or as inherited from a formula as follows.

#### *Correctly Rounded*

Operations described as “correctly rounded” will return the infinitely precise result,  $x$ , rounded so as to be representable in floating-point. The rounding mode is not specified, unless the entry point is declared with the `RoundingModeRTE` or the `RoundingModeRTZ` execution mode. These execution modes affect only correctly rounded SPIR-V instructions. These execution modes do not affect `OpQuantizeToF16`. If the rounding mode is not specified then this rounding is implementation specific, subject to the following rules. If  $x$  is exactly representable then  $x$  will be returned. Otherwise, either the floating-point value closest to and no less than  $x$  or the value closest to and no greater than  $x$  will be returned.

## ULP

Where an error bound of n ULP (units in the last place) is given, for an operation with infinitely precise result x the value returned **must** be in the range  $[x - n * \text{ulp}(x), x + n * \text{ulp}(x)]$ . The function  $\text{ulp}(x)$  is defined as follows:

If there exist non-equal floating-point numbers a and b such that  $a \leq x \leq b$  then  $\text{ulp}(x)$  is the minimum possible distance between such numbers,  $\text{ulp}(x) = \min_{a, b} |b - a|$ . If such numbers do not exist then  $\text{ulp}(x)$  is defined to be the difference between the two finite floating-point numbers nearest to x.

Where the range of allowed return values includes any value of magnitude larger than that of the largest representable finite floating-point number, operations **may**, additionally, return either an infinity of the appropriate sign or the finite number with the largest magnitude of the appropriate sign. If the infinitely precise result of the operation is not mathematically defined then the value returned is undefined.

## Inherited From ...

Where an operation's precision is described as being inherited from a formula, the result returned must be at least as accurate as the result of computing an approximation to x using a formula equivalent to the given formula applied to the supplied inputs. Specifically, the formula given may be transformed using the mathematical associativity, commutativity and distributivity of the operators involved to yield an equivalent formula. The SPIR-V precision rules, when applied to each such formula and the given input values, define a range of permitted values. If NaN is one of the permitted values then the operation may return any result, otherwise let the largest permitted value in any of the ranges be  $F_{\max}$  and the smallest be  $F_{\min}$ . The operation must return a value in the range  $[x - E, x + E]$  where  $E = \max(|x - F_{\min}|, |x - F_{\max}|)$ . If the entry point is declared with the **DenormFlushToZero** execution mode, then any intermediate denormal value(s) while evaluating the formula **may** be flushed to zero. Denormal final results **must** be flushed to zero. If the entry point is declared with the **DenormPreserve** execution mode, then denormals **must** be preserved throughout the formula.

For half- (16 bit) and single- (32 bit) precision instructions, precisions are **required** to be at least as follows:

Table 76. Precision of core SPIR-V Instructions

Instruction	Single precision, unless decorated with <b>RelaxedPrecision</b>	Half precision
<b>OpFAdd</b>	Correctly rounded.	
<b>OpFSub</b>	Correctly rounded.	
<b>OpFMul</b> , <b>OpVectorTimesScalar</b> , <b>OpMatrixTimesScalar</b>	Correctly rounded.	
<b>OpDot</b> (x, y)	Inherited from $\sum_{i=0}^{n-1} x_i \times y_i$ .	
<b>OpFOrdEqual</b> , <b>OpFUnordEqual</b>	Correct result.	
<b>OpFOrdLessThan</b> , <b>OpFUnordLessThan</b>	Correct result.	

Instruction	Single precision, unless decorated with <code>RelaxedPrecision</code>	Half precision
<code>OpFOrdGreaterThan</code> , <code>OpFunordGreaterThan</code>	Correct result.	
<code>OpFOrdLessThanEqual</code> , <code>OpFunordLessThanEqual</code>	Correct result.	
<code>OpFOrdGreaterThanOrEqual</code> , <code>OpFunordGreaterThanOrEqual</code>	Correct result.	
<code>OpFDiv(x,y)</code>	2.5 ULP for $ y $ in the range $[2^{-126}, 2^{126}]$ .	2.5 ULP for $ y $ in the range $[2^{-14}, 2^{14}]$ .
<code>OpFRem(x,y)</code>	Inherited from $x - y \times \text{trunc}(x/y)$ .	
<code>OpFMod(x,y)</code>	Inherited from $x - y \times \text{floor}(x/y)$ .	
conversions between types	Correctly rounded.	

*Note*



The `OpFRem` and `OpFMod` instructions use cheap approximations of remainder, and the error can be large due to the discontinuity in `trunc()` and `floor()`. This can produce mathematically unexpected results in some cases, such as `FMod(x,x)` computing  $x$  rather than 0, and can also cause the result to have a different sign than the infinitely precise result.

Table 77. Precision of GLSL.std.450 Instructions

Instruction	Single precision, unless decorated with <code>RelaxedPrecision</code>	Half precision
<code>fma()</code>	Inherited from <code>OpFMul</code> followed by <code>OpFAdd</code> .	
<code>exp(x)</code> , <code>exp2(x)</code>	$3 + 2 \times  x $ ULP.	$1 + 2 \times  x $ ULP.
<code>log()</code> , <code>log2()</code>	3 ULP outside the range $[0.5, 2.0]$ . Absolute error $< 2^{-21}$ inside the range $[0.5, 2.0]$ .	3 ULP outside the range $[0.5, 2.0]$ . Absolute error $< 2^{-7}$ inside the range $[0.5, 2.0]$ .
<code>pow(x, y)</code>	Inherited from <code>exp2(y × log2(x))</code> .	
<code>sqrt()</code>	Inherited from <code>1.0 / inversesqrt()</code> .	
<code>inversesqrt()</code>	2 ULP.	
<code>radians(x)</code>	Inherited from $x \times \frac{\pi}{180}$ .	
<code>degrees(x)</code>	Inherited from $x \times \frac{180}{\pi}$ .	
<code>sin()</code>	Absolute error $\leq 2^{-11}$ inside the range $[-\pi, \pi]$ .	Absolute error $\leq 2^{-7}$ inside the range $[-\pi, \pi]$ .

<b>Instruction</b>	<b>Single precision, unless decorated with RelaxedPrecision</b>	<b>Half precision</b>
<code>cos()</code>	Absolute error $\leq 2^{-11}$ inside the range $[-\pi, \pi]$ .	Absolute error $\leq 2^{-7}$ inside the range $[-\pi, \pi]$ .
<code>tan()</code>	Inherited from $\frac{\sin()}{\cos()}$ .	
<code>asin(x)</code>	Inherited from $\text{atan2}(x, \sqrt{1.0 - x \times x})$ .	
<code>acos(x)</code>	Inherited from $\text{atan2}(\sqrt{1.0 - x \times x}, x)$ .	
<code>atan(), atan2()</code>	4096 ULP	5 ULP.
<code>sinh(x)</code>	Inherited from $(\exp(x) - \exp(-x)) \times 0.5$ .	
<code>cosh(x)</code>	Inherited from $(\exp(x) + \exp(-x)) \times 0.5$ .	
<code>tanh()</code>	Inherited from $\frac{\sinh()}{\cosh()}$ .	
<code>asinh(x)</code>	Inherited from $\log(x + \sqrt{x \times x + 1.0})$ .	
<code>acosh(x)</code>	Inherited from $\log(x + \sqrt{x \times x - 1.0})$ .	
<code>atanh(x)</code>	Inherited from $\log(\frac{1.0 + x}{1.0 - x}) \times 0.5$ .	
<code>frexp()</code>	Correctly rounded.	
<code>ldexp()</code>	Correctly rounded.	
<code>length(x)</code>	Inherited from $\sqrt(\text{dot}(x, x))$ .	
<code>distance(x, y)</code>	Inherited from $\text{length}(x - y)$ .	
<code>cross()</code>	Inherited from <code>OpFSub(OpFMul, OpFMul)</code> .	
<code>normalize(x)</code>	Inherited from $\frac{x}{\text{length}(x)}$ .	
<code>faceforward(N, I, NRef)</code>	Inherited from $\text{dot}(N\text{Ref}, I) < 0.0 ? N : -N$ .	
<code>reflect(x, y)</code>	Inherited from $x - 2.0 \times \text{dot}(y, x) \times y$ .	
<code>refract(I, N, eta)</code>	Inherited from $k < 0.0 ? 0.0 : \text{eta} \times I - (\text{eta} \times \text{dot}(N, I) + \sqrt{k}) \times N$ , where $k = 1 - \text{eta} \times \text{eta} \times (1.0 - \text{dot}(N, I) \times \text{dot}(N, I))$ .	
<code>round</code>	Correctly rounded.	
<code>roundEven</code>	Correctly rounded.	
<code>trunc</code>	Correctly rounded.	
<code>fabs</code>	Correctly rounded.	

Instruction	Single precision, unless decorated with <code>RelaxedPrecision</code>	Half precision
<code>fsign</code>	Correctly rounded.	
<code>floor</code>	Correctly rounded.	
<code>ceil</code>	Correctly rounded.	
<code>fract</code>	Correctly rounded.	
<code>modf</code>	Correctly rounded.	
<code>fmin</code>	Correctly rounded.	
<code>fmax</code>	Correctly rounded.	
<code>fclamp</code>	Correctly rounded.	
<code>fmix(x, y, a)</code>	Inherited from $x \times (1.0 - a) + y \times a$ .	
<code>step</code>	Correctly rounded.	
<code>smoothStep(edge0, edge1, x)</code>	Inherited from $t \times t \times (3.0 - 2.0 \times t)$ , where $t = clamp(\frac{x - edge0}{edge1 - edge0}, 0.0, 1.0)$ .	
<code>nmin</code>	Correctly rounded.	
<code>nmax</code>	Correctly rounded.	
<code>nclamp</code>	Correctly rounded.	

GLSL.std.450 extended instructions specifically defined in terms of the above instructions inherit the above errors. GLSL.std.450 extended instructions not listed above and not defined in terms of the above have undefined precision.

For the `OpSRem` and `OpSMod` instructions, if either operand is negative the result is undefined.

*Note*



While the `OpSRem` and `OpSMod` instructions are supported by the Vulkan environment, they require non-negative values and thus do not enable additional functionality beyond what `OpUMod` provides.

`OpCooperativeMatrixMulAddNV` performs its operations in an implementation-dependent order and internal precision.

## Compatibility Between SPIR-V Image Formats And Vulkan Formats

Images which are read from or written to by shaders **must** have SPIR-V image formats compatible with the Vulkan image formats backing the image under the circumstances described for [texture image validation](#). The compatible formats are:

*Table 78. SPIR-V and Vulkan Image Format Compatibility*

<b>SPIR-V Image Format</b>	<b>Compatible Vulkan Format</b>
Rgba32f	VK_FORMAT_R32G32B32A32_SFLOAT
Rgba16f	VK_FORMAT_R16G16B16A16_SFLOAT
R32f	VK_FORMAT_R32_SFLOAT
Rgba8	VK_FORMAT_R8G8B8A8_UNORM
Rgba8Snorm	VK_FORMAT_R8G8B8A8_SNORM
Rg32f	VK_FORMAT_R32G32_SFLOAT
Rg16f	VK_FORMAT_R16G16_SFLOAT
R11fG11fB10f	VK_FORMAT_B10G11R11_UFLOAT_PACK32
R16f	VK_FORMAT_R16_SFLOAT
Rgba16	VK_FORMAT_R16G16B16A16_UNORM
Rgb10A2	VK_FORMAT_A2B10G10R10_UNORM_PACK32
Rg16	VK_FORMAT_R16G16_UNORM
Rg8	VK_FORMAT_R8G8_UNORM
R16	VK_FORMAT_R16_UNORM
R8	VK_FORMAT_R8_UNORM
Rgba16Snorm	VK_FORMAT_R16G16B16A16_SNORM
Rg16Snorm	VK_FORMAT_R16G16_SNORM
Rg8Snorm	VK_FORMAT_R8G8_SNORM
R16Snorm	VK_FORMAT_R16_SNORM
R8Snorm	VK_FORMAT_R8_SNORM
Rgba32i	VK_FORMAT_R32G32B32A32_SINT
Rgba16i	VK_FORMAT_R16G16B16A16_SINT
Rgba8i	VK_FORMAT_R8G8B8A8_SINT
R32i	VK_FORMAT_R32_SINT
Rg32i	VK_FORMAT_R32G32_SINT
Rg16i	VK_FORMAT_R16G16_SINT
Rg8i	VK_FORMAT_R8G8_SINT
R16i	VK_FORMAT_R16_SINT
R8i	VK_FORMAT_R8_SINT
Rgba32ui	VK_FORMAT_R32G32B32A32_UINT
Rgba16ui	VK_FORMAT_R16G16B16A16_UINT
Rgba8ui	VK_FORMAT_R8G8B8A8_UINT
R32ui	VK_FORMAT_R32_UINT
Rgb10a2ui	VK_FORMAT_A2B10G10R10_UINT_PACK32
Rg32ui	VK_FORMAT_R32G32_UINT
Rg16ui	VK_FORMAT_R16G16_UINT
Rg8ui	VK_FORMAT_R8G8_UINT
R16ui	VK_FORMAT_R16_UINT
R8ui	VK_FORMAT_R8_UINT

# Appendix B: Memory Model

## Agent

*Operation* is a general term for any task that is executed on the system.



An operation is by definition something that is executed, thus if an instruction is skipped due to flow control it does not constitute an operation.

Each operation is executed by a particular *agent*. Possible agents include each shader invocation, each host thread, and each fixed-function stage of the pipeline.

## Memory Location

A *memory location* identifies unique storage for 8 bits of data. Memory operations access a *set of memory locations* consisting of one or more memory locations at a time, e.g. an operation accessing a 32-bit integer in memory would read/write a set of four memory locations. Memory operations that access whole aggregates **may** access any padding bytes between elements or members, but no padding bytes at the end of the aggregate. Two sets of memory locations *overlap* if the intersection of their sets of memory locations is non-empty. A memory operation **must** not affect memory at a memory location not within its set of memory locations.

Memory locations for buffers and images are explicitly allocated in `VkDeviceMemory` objects, and are implicitly allocated for SPIR-V variables in each shader invocation.

## Allocation

The values stored in newly allocated memory locations are determined by a SPIR-V variable's initializer, if present, or else are undefined. At the time an allocation is created there have been no **memory operations** to any of its memory locations. The initialization is not considered to be a memory operation.



For tessellation control shader output variables, a consequence of initialization not being considered a memory operation is that some implementations may need to insert a barrier between the initialization of the output variables and any reads of those variables.

## Memory Operation

For an operation A and memory location M:

- A *reads* M if and only if the data stored in M is an input to A.
- A *writes* M if and only if the data output from A is stored to M.
- A *accesses* M if and only if it either reads or writes (or both) M.



A write whose value is the same as what was already in those memory locations is still considered to be a write and has all the same effects.

## Reference

A *reference* is an object that a particular agent **can** use to access a set of memory locations. On the host, a reference is a host virtual address. On the device, a reference is:

- The descriptor that a variable is bound to, for variables in Image, Uniform, or StorageBuffer storage classes. If the variable is an array (or array of arrays, etc.) then each element of the array **may** be a unique reference.
- The address range for a buffer in `PhysicalStorageBuffer` storage class, where the base of the address range is queried with `vkGetBufferDeviceAddressKHR` and the length of the range is the size of the buffer.
- The variable itself for variables in other storage classes.

Two memory accesses through distinct references **may** require availability and visibility operations as defined [below](#).

## Program-Order

A *dynamic instance* of an instruction is defined in SPIR-V (<https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html#DynamicInstance>) as a way of referring to a particular execution of a static instruction. Program-order is an ordering on dynamic instances of instructions executed by a single shader invocation:

- (Basic block): If instructions A and B are in the same basic block, and A is listed in the module before B, then the n'th dynamic instance of A is program-ordered before the n'th dynamic instance of B.
- (Branch): The dynamic instance of a branch or switch instruction is program-ordered before the dynamic instance of the OpLabel instruction to which it transfers control.
- (Call entry): The dynamic instance of a function call instruction is program-ordered before the dynamic instances of the `OpFunctionParameter` instructions and the body of the called function.
- (Call exit): The dynamic instance of the instruction following a function call instruction is program-ordered after the dynamic instance of the return instruction executed by the called function.
- (Transitive Closure): If dynamic instance A of any instruction is program-ordered before dynamic instance B of any instruction and B is program-ordered before dynamic instance C of any instruction then A is program-ordered before C.
- (Complete definition): No other dynamic instances are program-ordered.

For instructions executed on the host, the source language defines the program-order relation (e.g. as “sequenced-before”).

# Scope

A *scope* describes a set of shader invocations, where each such set is a *scope instance*. Scopes are defined hierarchically such that a more inclusive scope includes one or more sets of less inclusive scope instances. The scopes defined by SPIR-V are as follows, defined from most inclusive to least inclusive:

- **CrossDevice** identifies all shader invocations in a Vulkan instance across all shader launches, and all host threads interacting with that instance.
- **Device** identifies all shader invocations that execute on a given device, including those from different shader launches.
- **QueueFamilyKHR** identifies all shader invocations that execute on any queue in a given queue family, including those from different shader launches.
- **FragmentInterlock** identifies sets of fragment shader invocations that overlap as defined in [Fragment Shader Interlock](#). This scope does not exist as an enum in SPIR-V, it is only implicitly used as a memory scope by **OpBeginInvocationInterlockEXT** and **OpEndInvocationInterlockEXT**.
- **Workgroup** identifies all invocations in a single workgroup.
- **Subgroup** identifies all invocations in a single subgroup.
- **Invocation** identifies a single invocation.

Atomic and barrier instructions include scopes which identify sets of shader invocations that **must** obey the requested ordering and atomicity rules of the operation, as defined below.

# Atomic Operation

An *atomic operation* on the device is any SPIR-V operation whose name begins with **OpAtomic**. An atomic operation on the host is any operation performed with an `std::atomic` typed object.

Each atomic operation has a memory *scope* and a *semantics*. Informally, the scope determines which other agents it is atomic with respect to, and the *semantics* constrains its ordering against other memory accesses. Device atomic operations have explicit scopes and semantics. Each host atomic operation implicitly uses the **CrossDevice** scope, and uses a memory semantics equivalent to a C++ `std::memory_order` value of relaxed, acquire, release, acq\_rel, or seq\_cst.

Two atomic operations A and B are *potentially-mutually-ordered* if and only if all of the following are true:

- They access the same set of memory locations.
- They use the same reference.
- A is in the instance of B's memory scope.
- B is in the instance of A's memory scope.
- A and B are not the same operation (irreflexive).

Two atomic operations A and B are *mutually-ordered* if and only if they are potentially-mutually-ordered and any of the following are true:

- A and B are both device operations.
- A and B are both host operations.
- A is a device operation, B is a host operation, and the implementation supports concurrent host- and device-atomics.



If two atomic operations are not mutually-ordered, and if their sets of memory locations overlap, then each **must** be synchronized against the other as if they were non-atomic operations.

## Scoped Modification Order

For a given atomic write A, all atomic writes that are mutually-ordered with A occur in an order known as A's *scoped modification order*. A's scoped modification order relates no other operations.



Invocations outside the instance of A's memory scope **may** observe the values at A's set of memory locations becoming visible to it in an order that disagrees with the scoped modification order.



It is valid to have non-atomic operations or atomics in a different scope instance to the same set of memory locations, as long as they are synchronized against each other as if they were non-atomic (if they are not, it is treated as a [data race](#)). That means this definition of A's scoped modification order could include atomic operations that occur much later, after intervening non-atomics. That is a bit non-intuitive, but it helps to keep this definition simple and non-circular.

## Memory Semantics

Non-atomic memory operations, by default, **may** be observed by one agent in a different order than they were written by another agent.

Atomics and some synchronization operations include *memory semantics*, which are flags that constrain the order in which other memory accesses (including non-atomic memory accesses and [availability and visibility operations](#)) performed by the same agent **can** be observed by other agents, or **can** observe accesses by other agents.

Device instructions that include semantics are [OpAtomic\\*](#), [OpControlBarrier](#), [OpMemoryBarrier](#), and [OpMemoryNamedBarrier](#). Host instructions that include semantics are some std::atomic methods and memory fences.

SPIR-V supports the following memory semantics:

- Relaxed: No constraints on order of other memory accesses.
- Acquire: A memory read with this semantic performs an *acquire operation*. A memory barrier with this semantic is an *acquire barrier*.
- Release: A memory write with this semantic performs a *release operation*. A memory barrier with this semantic is a *release barrier*.

- **AcquireRelease:** A memory read-modify-write operation with this semantic performs both an acquire operation and a release operation, and inherits the limitations on ordering from both of those operations. A memory barrier with this semantic is both a release and acquire barrier.



SPIR-V does not support “consume” semantics on the device.

The memory semantics operand also includes *storage class semantics* which indicate which storage classes are constrained by the synchronization. SPIR-V storage class semantics include:

- UniformMemory
- WorkgroupMemory
- ImageMemory
- OutputMemoryKHR

Each SPIR-V memory operation accesses a single storage class. Semantics in synchronization operations can include a combination of storage classes.

The UniformMemory storage class semantic applies to accesses to memory in the PhysicalStorageBuffer, Uniform and StorageBuffer storage classes. The WorkgroupMemory storage class semantic applies to accesses to memory in the Workgroup storage class. The ImageMemory storage class semantic applies to accesses to memory in the Image storage class. The OutputMemoryKHR storage class semantic applies to accesses to memory in the Output storage class.



Informally, these constraints limit how memory operations can be reordered, and these limits apply not only to the order of accesses as performed in the agent that executes the instruction, but also to the order the effects of writes become visible to all other agents within the same instance of the instruction’s memory scope.



Release and acquire operations in different threads **can** act as synchronization operations, to guarantee that writes that happened before the release are visible after the acquire. (This is not a formal definition, just an informative forward reference.)



The OutputMemoryKHR storage class semantic is only useful in tessellation control shaders, which is the only execution model where output variables are shared between invocations.

The memory semantics operand also optionally includes availability and visibility flags, which apply optional availability and visibility operations as described in [availability and visibility](#). The availability/visibility flags are:

- **MakeAvailable:** Semantics **must** be Release or AcquireRelease. Performs an availability operation before the release operation or barrier.
- **MakeVisible:** Semantics **must** be Acquire or AcquireRelease. Performs a visibility operation after the acquire operation or barrier.

The specifics of these operations are defined in [Availability and Visibility Semantics](#).

Host atomic operations **may** support a different list of memory semantics and synchronization operations, depending on the host architecture and source language.

## Release Sequence

After an atomic operation A performs a release operation on a set of memory locations M, the *release sequence headed by A* is the longest continuous subsequence of A's scoped modification order that consists of:

- the atomic operation A as its first element
- atomic read-modify-write operations on M by any agent



The atomics in the last bullet **must** be mutually-ordered with A by virtue of being in A's scoped modification order.



This intentionally omits “atomic writes to M performed by the same agent that performed A”, which is present in the corresponding C++ definition.

## Synchronizes-With

*Synchronizes-with* is a relation between operations, where each operation is either an atomic operation or a memory barrier (aka fence on the host).

If A and B are atomic operations, then A synchronizes-with B if and only if all of the following are true:

- A performs a release operation
- B performs an acquire operation
- A and B are mutually-ordered
- B reads a value written by A or by an operation in the release sequence headed by A

`OpControlBarrier`, `OpMemoryBarrier`, and `OpMemoryNamedBarrier` are *memory barrier* instructions in SPIR-V.

If A is a release barrier and B is an atomic operation that performs an acquire operation, then A synchronizes-with B if and only if all of the following are true:

- there exists an atomic write X (with any memory semantics)
- A is program-ordered before X
- X and B are mutually-ordered
- B reads a value written by X or by an operation in the release sequence headed by X
  - If X is relaxed, it is still considered to head a hypothetical release sequence for this rule
- A and B are in the instance of each other's memory scopes

- X's storage class is in A's semantics.

If A is an atomic operation that performs a release operation and B is an acquire barrier, then A synchronizes-with B if and only if all of the following are true:

- there exists an atomic read X (with any memory semantics)
- X is program-ordered before B
- X and A are mutually-ordered
- X reads a value written by A or by an operation in the release sequence headed by A
- A and B are in the instance of each other's memory scopes
- X's storage class is in B's semantics.

If A is a release barrier and B is an acquire barrier, then A synchronizes-with B if all of the following are true:

- there exists an atomic write X (with any memory semantics)
- A is program-ordered before X
- there exists an atomic read Y (with any memory semantics)
- Y is program-ordered before B
- X and Y are mutually-ordered
- Y reads the value written by X or by an operation in the release sequence headed by X
  - If X is relaxed, it is still considered to head a hypothetical release sequence for this rule
- A and B are in the instance of each other's memory scopes
- X's and Y's storage class is in A's and B's semantics.
  - NOTE: X and Y must have the same storage class, because they are mutually ordered.

If A is a release barrier and B is an acquire barrier and C is a control barrier (where A can optionally equal C and B can optionally equal C), then A synchronizes-with B if all of the following are true:

- A is program-ordered before (or equals) C
- C is program-ordered before (or equals) B
- A and B are in the instance of each other's memory scopes
- A and B are in the instance of C's execution scope



This is similar to the barrier-barrier synchronization above, but with a control barrier filling the role of the relaxed atomics.

Let F be an ordering of fragment shader invocations, such that invocation  $F_1$  is ordered before invocation  $F_2$  if and only if  $F_1$  and  $F_2$  overlap as described in [Fragment Shader Interlock](#) and  $F_1$  executes the interlocked code before  $F_2$ .

If A is an `OpEndInvocationInterlockEXT` instruction and B is an `OpBeginInvocationInterlockEXT`

instruction, then A synchronizes-with B if the agent that executes A is ordered before the agent that executes B in F. A and B are both considered to have [FragmentInterlock](#) memory scope and semantics of UniformMemory and ImageMemory, and A is considered to have Release semantics and B is considered to have Acquire semantics.

No other release and acquire barriers synchronize-with each other.

## System-Synchronizes-With

*System-synchronizes-with* is a relation between arbitrary operations on the device or host. Certain operations system-synchronize-with each other, which informally means the first operation occurs before the second and that the synchronization is performed without using application-visible memory accesses.

If there is an [execution dependency](#) between two operations A and B, then the operation in the first synchronization scope system-synchronizes-with the operation in the second synchronization scope.



This covers all Vulkan synchronization primitives, including device operations executing before a synchronization primitive is signaled, wait operations happening before subsequent device operations, signal operations happening before host operations that wait on them, and host operations happening before [vkQueueSubmit](#). The list is spread throughout the synchronization chapter, and is not repeated here.

System-synchronizes-with implicitly includes all storage class semantics and has [CrossDevice](#) scope.

If A system-synchronizes-with B, we also say A is *system-synchronized-before* B and B is *system-synchronized-after* A.

## Private vs. Non-Private

By default, non-atomic memory operations are treated as *private*, meaning such a memory operation is not intended to be used for communication with other agents. Memory operations with the NonPrivatePointerKHR/NonPrivateTexelKHR bit set are treated as *non-private*, and are intended to be used for communication with other agents.

More precisely, for private memory operations to be [Location-Ordered](#) between distinct agents requires using system-synchronizes-with rather than shader-based synchronization. Non-private memory operations still obey program-order.

Atomic operations are always considered non-private.

## Inter-Thread-Happens-Before

Let SC be a non-empty set of storage class semantics. Then (using template syntax) operation A *inter-thread-happens-before*<SC> operation B if and only if any of the following is true:

- A system-synchronizes-with B
- A synchronizes-with B, and both A and B have all of SC in their semantics
- A is an operation on memory in a storage class in SC or that has all of SC in its semantics, B is a release barrier or release atomic with all of SC in its semantics, and A is program-ordered before B
- A is an acquire barrier or acquire atomic with all of SC in its semantics, B is an operation on memory in a storage class in SC or that has all of SC in its semantics, and A is program-ordered before B
- A and B are both host operations and A inter-thread-happens-before B as defined in the host language spec
- A inter-thread-happens-before $\langle SC \rangle$  some X and X inter-thread-happens-before $\langle SC \rangle$  B

## Happens-Before

Operation A *happens-before* operation B if and only if any of the following is true:

- A is program-ordered before B
- A inter-thread-happens-before $\langle SC \rangle$  B for some set of storage classes SC

*Happens-after* is defined similarly.



Unlike C++, happens-before is not always sufficient for a write to be visible to a read. Additional [availability and visibility](#) operations **may** be required for writes to be [visible-to](#) other memory accesses.



Happens-before is not transitive, but each of program-order and inter-thread-happens-before $\langle SC \rangle$  are transitive. These can be thought of as covering the “single-threaded” case and the “multi-threaded” case, and it is not necessary (and not valid) to form chains between the two.

## Availability and Visibility

*Availability* and *visibility* are states of a write operation, which (informally) track how far the write has permeated the system, i.e. which agents and references are able to observe the write. Availability state is per *memory domain*. Visibility state is per (agent,reference) pair. Availability and visibility states are per-memory location for each write.

Memory domains are named according to the agents whose memory accesses use the domain. Domains used by shader invocations are organized hierarchically into multiple smaller memory domains which correspond to the different [scopes](#). The memory domains defined in Vulkan include:

- *host* - accessible by host agents
- *device* - accessible by all device agents for a particular device

- *shader* - accessible by shader agents for a particular device, corresponding to the [Device](#) scope
- *queue family instance* - accessible by shader agents in a single queue family, corresponding to the [QueueFamilyKHR](#) scope.
- *fragment interlock instance* - accessible by fragment shader agents that [overlap](#), corresponding to the [FragmentInterlock](#) scope.
- *workgroup instance* - accessible by shader agents in the same workgroup, corresponding to the [Workgroup](#) scope.
- *subgroup instance* - accessible by shader agents in the same subgroup, corresponding to the [Subgroup](#) scope.

 These do not correspond to storage classes or device-local and host-local [VkDeviceMemory](#) allocations, rather they indicate whether a write can be made visible only to agents in the same subgroup, same workgroup, overlapping fragment shader invocation, in any shader invocation, or anywhere on the device, or host. The shader, queue family instance, fragment interlock instance, workgroup instance, and subgroup instance domains are only used for shader-based availability/visibility operations, in other cases writes can be made available from/visible to the shader via the device domain.

*Availability operations, visibility operations, and memory domain operations* alter the state of the write operations that happen-before them, and which are included in their *source scope* to be available or visible to their *destination scope*.

- For an availability operation, the source scope is a set of (agent,reference,memory location) tuples, and the destination scope is a set of memory domains.
- For a memory domain operation, the source scope is a memory domain and the destination scope is a memory domain.
- For a visibility operation, the source scope is a set of memory domains and the destination scope is a set of (agent,reference,memory location) tuples.

How the scopes are determined depends on the specific operation. Availability and memory domain operations expand the set of memory domains to which the write is available. Visibility operations expand the set of (agent,reference,memory location) tuples to which the write is visible.

Recall that availability and visibility states are per-memory location, and let W be a write operation to one or more locations performed by agent A via reference R. Let L be one of the locations written. (W,L) (the write W to L), is initially not available to any memory domain and only visible to (A,R,L). An availability operation AV that happens-after W and that includes (A,R,L) in its source scope makes (W,L) *available* to the memory domains in its destination scope.

A memory domain operation DOM that happens-after AV and for which (W,L) is available in the source scope makes (W,L) available in the destination memory domain.

A visibility operation VIS that happens-after AV (or DOM) and for which (W,L) is available in any domain in the source scope makes (W,L) *visible* to all (agent,reference,L) tuples included in its destination scope.

If write  $W_2$  happens-after  $W$ , and their sets of memory locations overlap, then  $W$  will not be available/visible to all agents/references for those memory locations that overlap (and future AV/DOM/VIS ops cannot revive  $W$ 's write to those locations).

Availability, memory domain, and visibility operations are treated like other non-atomic memory accesses for the purpose of [memory semantics](#), meaning they can be ordered by release-acquire sequences or memory barriers.

An *availability chain* is a sequence of availability operations of increasing scope where element  $N+1$  of the chain is performed in the same scope instance as the destination of element  $N$  and element  $N$  happens-before element  $N+1$ . An example is an availability operation with destination scope of the workgroup instance domain that happens before an availability operation to the shader domain performed by an invocation in the same workgroup. An availability chain AVC that happens-after  $W$  and that includes (A,R,L) in the source scope makes (W,L) *available* to the memory domains in its final destination scope. An availability chain with a single element is just the availability operation.

Similarly, a *visibility chain* is a sequence of visibility operations of decreasing scope where element  $N$  of the chain is performed in the same scope instance as the source of element  $N+1$  and element  $N$  happens-before element  $N+1$ . An example is a visibility operation with source scope of the shader domain that happens before a visibility operation with source scope of the workgroup instance domain performance by an invocation in the same workgroup. A visibility chain VISC that happens-after AVC (or DOM) and for which (W,L) is available in any domain in the source scope makes (W,L) *visible* to all (agent,reference,L) tuples included in its final destination scope. A visibility chain with a single element is just the visibility operation.

## Availability, Visibility, and Domain Operations

The following operations generate availability, visibility, and domain operations. When multiple availability/visibility/domain operations are described, they are system-synchronized-with each other in the order listed.

An operation that performs a [memory dependency](#) generates:

- If the source access mask includes `VK_ACCESS_HOST_WRITE_BIT`, then the dependency includes a memory domain operation from host domain to device domain.
- An availability operation with source scope of all writes in the first [access scope](#) of the dependency and a destination scope of the device domain.
- A visibility operation with source scope of the device domain and destination scope of the second access scope of the dependency.
- If the destination access mask includes `VK_ACCESS_HOST_READ_BIT` or `VK_ACCESS_HOST_WRITE_BIT`, then the dependency includes a memory domain operation from device domain to host domain.

[vkFlushMappedMemoryRanges](#) performs an availability operation, with a source scope of (agents,references) = (all host threads, all mapped memory ranges passed to the command), and destination scope of the host domain.

[vkInvalidateMappedMemoryRanges](#) performs a visibility operation, with a source scope of the host domain and a destination scope of (agents,references) = (all host threads, all mapped memory

ranges passed to the command).

`vkQueueSubmit` performs a memory domain operation from host to device, and a visibility operation with source scope of the device domain and destination scope of all agents and references on the device.

## Availability and Visibility Semantics

A memory barrier or atomic operation via agent A that includes `MakeAvailable` in its semantics performs an availability operation whose source scope includes agent A and all references in the storage classes in that instruction's storage class semantics, and all memory locations, and whose destination scope is a set of memory domains selected as specified below. The implicit availability operation is program-ordered between the barrier or atomic and all other operations program-ordered before the barrier or atomic.

A memory barrier or atomic operation via agent A that includes `MakeVisible` in its semantics performs a visibility operation whose source scope is a set of memory domains selected as specified below, and whose destination scope includes agent A and all references in the storage classes in that instruction's storage class semantics, and all memory locations. The implicit visibility operation is program-ordered between the barrier or atomic and all other operations program-ordered after the barrier or atomic.

The memory domains are selected based on the memory scope of the instruction as follows:

- `Device` scope uses the shader domain
- `QueueFamilyKHR` scope uses the queue family instance domain
- `FragmentInterlock` scope uses the fragment interlock instance domain
- `Workgroup` scope uses the workgroup instance domain
- `Subgroup` uses the subgroup instance domain
- `Invocation` perform no availability/visibility operations.

When an availability operation performed by an agent A includes a memory domain D in its destination scope, where D corresponds to scope instance S, it also includes the memory domains that correspond to each smaller scope instance S' that is a subset of S and that includes A. Similarly for visibility operations.

## Per-Instruction Availability and Visibility Semantics

A memory write instruction that includes `MakePointerAvailable`, or an image write instruction that includes `MakeTexelAvailable`, performs an availability operation whose source scope includes the agent and reference used to perform the write and the memory locations written by the instruction, and whose destination scope is a set of memory domains selected by the Scope operand specified in [Availability and Visibility Semantics](#). The implicit availability operation is program-ordered between the write and all other operations program-ordered after the write.

A memory read instruction that includes `MakePointerVisible`, or an image read instruction that includes `MakeTexelVisible`, performs a visibility operation whose source scope is a set of memory

domains selected by the Scope operand as specified in [Availability and Visibility Semantics](#), and whose destination scope includes the agent and reference used to perform the read and the memory locations read by the instruction. The implicit visibility operation is program-ordered between read and all other operations program-ordered before the read.



Although reads with per-instruction visibility only perform visibility ops from the shader or fragment interlock instance or workgroup instance or subgroup instance domain, they will also see writes that were made visible via the device domain, i.e. those writes previously performed by non-shader agents and made visible via API commands.



It is expected that all invocations in a subgroup execute on the same processor with the same path to memory, and thus availability and visibility operations with subgroup scope can be expected to be “free”.

## Location-Ordered

Let X and Y be memory accesses to overlapping sets of memory locations M, where  $X \neq Y$ . Let  $(A_x, R_x)$  be the agent and reference used for X, and  $(A_y, R_y)$  be the agent and reference used for Y. For now, let “ $\rightarrow$ ” denote happens-before and “ $\rightarrow^{\text{rcpo}}$ ” denote the reflexive closure of program-ordered before.

If  $D_1$  and  $D_2$  are different memory domains, then let  $\text{DOM}(D_1, D_2)$  be a memory domain operation from  $D_1$  to  $D_2$ . Otherwise, let  $\text{DOM}(D, D)$  be a placeholder such that  $X \rightarrow \text{DOM}(D, D) \rightarrow Y$  if and only if  $X \rightarrow Y$ .

X is *location-ordered* before Y for a location L in M if and only if any of the following is true:

- $A_x == A_y$  and  $R_x == R_y$  and  $X \rightarrow Y$ 
  - NOTE: this case means no availability/visibility ops required when it is the same (agent,reference).
- X is a read, both X and Y are non-private, and  $X \rightarrow Y$
- X is a read, and X (transitively) system-synchronizes with Y
- If  $R_x == R_y$  and  $A_x$  and  $A_y$  access a common memory domain D (e.g. are in the same workgroup instance if D is the workgroup instance domain), and both X and Y are non-private:
  - X is a write, Y is a write,  $\text{AVC}(A_x, R_x, D, L)$  is an availability chain making  $(X, L)$  available to domain D, and  $X \rightarrow^{\text{rcpo}} \text{AVC}(A_x, R_x, D, L) \rightarrow Y$
  - X is a write, Y is a read,  $\text{AVC}(A_x, R_x, D, L)$  is an availability chain making  $(X, L)$  available to domain D,  $\text{VISC}(A_y, R_y, D, L)$  is a visibility chain making writes to L available in domain D visible to Y, and  $X \rightarrow^{\text{rcpo}} \text{AVC}(A_x, R_x, D, L) \rightarrow \text{VISC}(A_y, R_y, D, L) \rightarrow^{\text{rcpo}} Y$
  - If [VkPhysicalDeviceVulkanMemoryModelFeaturesKHR](#) `::vulkanMemoryModelAvailabilityVisibilityChains` is `VK_FALSE`, then AVC and VISC **must** each only have a single element in the chain, in each sub-bullet above.
- Let  $D_x$  and  $D_y$  each be either the device domain or the host domain, depending on whether  $A_x$  and  $A_y$  execute on the device or host:

- X is a write and Y is a write, and  $X \rightarrow AV(A_x, R_x, D_x, L) \rightarrow DOM(D_x, D_y) \rightarrow Y$
- X is a write and Y is a read, and  $X \rightarrow AV(A_x, R_x, D_x, L) \rightarrow DOM(D_x, D_y) \rightarrow VIS(A_y, R_y, D_y, L) \rightarrow Y$



The final bullet (synchronization through device/host domain) requires API-level synchronization operations, since the device/host domains are not accessible via shader instructions. And “device domain” is not to be confused with “device scope”, which synchronizes through the “shader domain”.

## Data Race

Let X and Y be operations that access overlapping sets of memory locations M, where  $X \neq Y$ , and at least one of X and Y is a write, and X and Y are not mutually-ordered atomic operations. If there does not exist a location-ordered relation between X and Y for each location in M, then there is a *data race*.

Applications **must** ensure that no data races occur during the execution of their application.



Data races can only occur due to instructions that are actually executed, and for example an instruction skipped due to flow control must not contribute to a data race.

## Visible-To

Let X be a write and Y be a read whose sets of memory locations overlap, and let M be the set of memory locations that overlap. Let  $M_2$  be a non-empty subset of M. Then X is *visible-to* Y for memory locations  $M_2$  if and only if all of the following are true:

- X is location-ordered before Y for each location L in  $M_2$ .
- There does not exist another write Z to any location L in  $M_2$  such that X is location-ordered before Z for location L and Z is location-ordered before Y for location L.

If X is visible-to Y, then Y reads the value written by X for locations  $M_2$ .



It is possible for there to be a write between X and Y that overwrites a subset of the memory locations, but the remaining memory locations ( $M_2$ ) will still be visible-to Y.

## Acyclicity

*Reads-from* is a relation between operations, where the first operation is a write, the second operation is a read, and the second operation reads the value written by the first operation. *From-reads* is a relation between operations, where the first operation is a read, the second operation is a write, and the first operation reads a value written earlier than the second operation in the second operation’s scoped modification order (or the first operation reads from the initial value, and the second operation is any write to the same locations).

Then the implementation **must** guarantee that no cycles exist in the union of the following relations:

- location-ordered
- scoped modification order (over all atomic writes)
- reads-from
- from-reads



This is a "consistency" axiom, which informally guarantees that sequences of operations can't violate causality.

## Scoped Modification Order Coherence

Let A and B be mutually-ordered atomic operations, where A is location-ordered before B. Then the following rules are a consequence of acyclicity:

- If A and B are both reads and A does not read the initial value, then the write that A takes its value from **must** be earlier in its own scoped modification order than (or the same as) the write that B takes its value from (no cycles between location-order, reads-from, and from-reads).
- If A is a read and B is a write and A does not read the initial value, then A **must** take its value from a write earlier than B in B's scoped modification order (no cycles between location-order, scope modification order, and reads-from).
- If A is a write and B is a read, then B **must** take its value from A or a write later than A in A's scoped modification order (no cycles between location-order, scoped modification order, and from-reads).
- If A and B are both writes, then A **must** be earlier than B in A's scoped modification order (no cycles between location-order and scoped modification order).
- If A is a write and B is a read-modify-write and B reads the value written by A, then B comes immediately after A in A's scoped modification order (no cycles between scoped modification order and from-reads).

## Shader I/O

If a shader invocation A in a shader stage other than **Vertex** performs a memory read operation X from an object in the **Input** storage class, then X is system-synchronized-after all writes to the corresponding **Output** storage variable(s) in the upstream shader invocation(s) that contribute to generating invocation A, and those writes are all visible-to X.



It is not necessary for the upstream shader invocations to have completed execution, they only need to have generated the output that is being read.

## Deallocation

A call to `vkFreeMemory` **must** happen-after all memory operations on all memory locations in that

[VkDeviceMemory](#) object.



Normally, device memory operations in a given queue are synchronized with [vkFreeMemory](#) by having a host thread wait on a fence signalled by that queue, and the wait happens-before the call to [vkFreeMemory](#) on the host.

The deallocation of SPIR-V variables is managed by the system and happens-after all operations on those variables.

## Informative Descriptions

This subsection is non-normative, and offers more easily understandable consequences of the memory model for app/compiler developers.

Let SC be the storage class(es) specified by a release or acquire operation or barrier.

- An atomic write with release semantics must not be reordered against any read or write to SC that is program-ordered before it (regardless of the storage class the atomic is in).
- An atomic read with acquire semantics must not be reordered against any read or write to SC that is program-ordered after it (regardless of the storage class the atomic is in).
- Any write to SC program-ordered after a release barrier must not be reordered against any read or write to SC program-ordered before that barrier.
- Any read from SC program-ordered before an acquire barrier must not be reordered against any read or write to SC program-ordered after the barrier.

A control barrier (even if it has no memory semantics) must not be reordered against any memory barriers.

This memory model allows memory accesses with and without availability and visibility operations, as well as atomic operations, all to be performed on the same memory location. This is critical to allow it to reason about memory that is reused in multiple ways, e.g. across the lifetime of different shader invocations or draw calls. While GLSL (and legacy SPIR-V) applies the “coherent” decoration to variables (for historical reasons), this model treats each memory access instruction as having optional implicit availability/visibility operations. GLSL to SPIR-V compilers should map all (non-atomic) operations on a coherent variable to `Make{Pointer, Texel}{Available}{Visible}` flags in this model.

Atomic operations implicitly have availability/visibility operations, and the scope of those operations is taken from the atomic operation’s scope.

## Tessellation Output Ordering

For SPIR-V that uses the Vulkan Memory Model, the [OutputMemory](#) storage class is used to synchronize accesses to tessellation control output variables. For legacy SPIR-V that does not enable the Vulkan Memory Model via [OpMemoryModel](#), tessellation outputs can be ordered using a control barrier with no particular memory scope or semantics, as defined below.

Let X and Y be memory operations performed by shader invocations  $A_x$  and  $A_y$ . Operation X is *tessellation-output-ordered* before operation Y if and only if all of the following are true:

- There is a dynamic instance of an `OpControlBarrier` instruction C such that X is program-ordered before C in  $A_x$  and C is program-ordered before Y in  $A_y$ .
- $A_x$  and  $A_y$  are in the same instance of C's execution scope.

If shader invocations  $A_x$  and  $A_y$  in the `TessellationControl` execution model execute memory operations X and Y, respectively, on the `Output` storage class, and X is tessellation-output-ordered before Y with a scope of `Workgroup`, then X is location-ordered before Y, and if X is a write and Y is a read then X is visible-to Y.

## Cooperative Matrix Memory Access

For each dynamic instance of a cooperative matrix load or store instruction (`OpCooperativeMatrixLoadNV` or `OpCooperativeMatrixStoreNV`), a single implementation-dependent invocation within the instance of the matrix's scope performs a non-atomic load or store (respectively) to each memory location that is defined to be accessed by the instruction.

# Appendix C: Compressed Image Formats

The compressed texture formats used by Vulkan are described in the specifically identified sections of the [Khronos Data Format Specification](#), version 1.1.

Unless otherwise described, the quantities encoded in these compressed formats are treated as normalized, unsigned values.

Those formats listed as sRGB-encoded have in-memory representations of R, G and B components which are nonlinearly-encoded as R', G', and B'; any alpha component is unchanged. As part of filtering, the nonlinear R', G', and B' values are converted to linear R, G, and B components; any alpha component is unchanged. The conversion between linear and nonlinear encoding is performed as described in the “KHR\_DF\_TRANSFER\_SRGB” section of the Khronos Data Format Specification.

# Block-Compressed Image Formats

Table 79. Mapping of Vulkan BC formats to descriptions

VkFormat	Khronos Data Format Specification description
Formats described in the “S3TC Compressed Texture Image Formats” chapter	
VK_FORMAT_BC1_RGB_UNORM_BLOCK	BC1 with no alpha
VK_FORMAT_BC1_RGB_SRGB_BLOCK	BC1 with no alpha, sRGB-encoded
VK_FORMAT_BC1_RGBA_UNORM_BLOCK	BC1 with alpha
VK_FORMAT_BC1_RGBA_SRGB_BLOCK	BC1 with alpha, sRGB-encoded
VK_FORMAT_BC2_UNORM_BLOCK	BC2
VK_FORMAT_BC2_SRGB_BLOCK	BC2, sRGB-encoded
VK_FORMAT_BC3_UNORM_BLOCK	BC3
VK_FORMAT_BC3_SRGB_BLOCK	BC3, sRGB-encoded
Formats described in the “RGTC Compressed Texture Image Formats” chapter	
VK_FORMAT_BC4_UNORM_BLOCK	BC4 unsigned
VK_FORMAT_BC4_SNORM_BLOCK	BC4 signed
VK_FORMAT_BC5_UNORM_BLOCK	BC5 unsigned
VK_FORMAT_BC5_SNORM_BLOCK	BC5 signed
Formats described in the “BPTC Compressed Texture Image Formats” chapter	
VK_FORMAT_BC6H_UFLOAT_BLOCK	BC6H (unsigned version)
VK_FORMAT_BC6H_SFLOAT_BLOCK	BC6H (signed version)
VK_FORMAT_BC7_UNORM_BLOCK	BC7
VK_FORMAT_BC7_SRGB_BLOCK	BC7, sRGB-encoded

# ETC Compressed Image Formats

The following formats are described in the “ETC2 Compressed Texture Image Formats” chapter of the [Khronos Data Format Specification](#).

*Table 80. Mapping of Vulkan ETC formats to descriptions*

VkFormat	Khronos Data Format Specification description
VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK	RGB ETC2
VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK	RGB ETC2 with sRGB encoding
VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK	RGB ETC2 with punch-through alpha
VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK	RGB ETC2 with punch-through alpha and sRGB
VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK	RGBA ETC2
VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK	RGBA ETC2 with sRGB encoding
VK_FORMAT_EAC_R11_UNORM_BLOCK	Unsigned R11 EAC
VK_FORMAT_EAC_R11_SNORM_BLOCK	Signed R11 EAC
VK_FORMAT_EAC_R11G11_UNORM_BLOCK	Unsigned RG11 EAC
VK_FORMAT_EAC_R11G11_SNORM_BLOCK	Signed RG11 EAC

# ASTC Compressed Image Formats

ASTC formats are described in the “ASTC Compressed Texture Image Formats” chapter of the [Khronos Data Format Specification](#).

*Table 81. Mapping of Vulkan ASTC formats to descriptions*

<b>VkFormat</b>	<b>Compressed texel block dimensions</b>	<b>sRGB-encoded</b>	<b>Profile</b>
VK_FORMAT_ASTC_4x4_UNORM_BLOCK	$4 \times 4$	No	LDR
VK_FORMAT_ASTC_4x4_SRGB_BLOCK	$4 \times 4$	Yes	LDR
VK_FORMAT_ASTC_5x4_UNORM_BLOCK	$5 \times 4$	No	LDR
VK_FORMAT_ASTC_5x4_SRGB_BLOCK	$5 \times 4$	Yes	LDR
VK_FORMAT_ASTC_5x5_UNORM_BLOCK	$5 \times 5$	No	LDR
VK_FORMAT_ASTC_5x5_SRGB_BLOCK	$5 \times 5$	Yes	LDR
VK_FORMAT_ASTC_6x5_UNORM_BLOCK	$6 \times 5$	No	LDR
VK_FORMAT_ASTC_6x5_SRGB_BLOCK	$6 \times 5$	Yes	LDR
VK_FORMAT_ASTC_6x6_UNORM_BLOCK	$6 \times 6$	No	LDR
VK_FORMAT_ASTC_6x6_SRGB_BLOCK	$6 \times 6$	Yes	LDR
VK_FORMAT_ASTC_8x5_UNORM_BLOCK	$8 \times 5$	No	LDR
VK_FORMAT_ASTC_8x5_SRGB_BLOCK	$8 \times 5$	Yes	LDR
VK_FORMAT_ASTC_8x6_UNORM_BLOCK	$8 \times 6$	No	LDR
VK_FORMAT_ASTC_8x6_SRGB_BLOCK	$8 \times 6$	Yes	LDR
VK_FORMAT_ASTC_8x8_UNORM_BLOCK	$8 \times 8$	No	LDR
VK_FORMAT_ASTC_8x8_SRGB_BLOCK	$8 \times 8$	Yes	LDR
VK_FORMAT_ASTC_10x5_UNORM_BLOCK	$10 \times 5$	No	LDR
VK_FORMAT_ASTC_10x5_SRGB_BLOCK	$10 \times 5$	Yes	LDR
VK_FORMAT_ASTC_10x6_UNORM_BLOCK	$10 \times 6$	No	LDR
VK_FORMAT_ASTC_10x6_SRGB_BLOCK	$10 \times 6$	Yes	LDR
VK_FORMAT_ASTC_10x8_UNORM_BLOCK	$10 \times 8$	No	LDR
VK_FORMAT_ASTC_10x8_SRGB_BLOCK	$10 \times 8$	Yes	LDR
VK_FORMAT_ASTC_10x10_UNORM_BLOCK	$10 \times 10$	No	LDR
VK_FORMAT_ASTC_10x10_SRGB_BLOCK	$10 \times 10$	Yes	LDR
VK_FORMAT_ASTC_12x10_UNORM_BLOCK	$12 \times 10$	No	LDR
VK_FORMAT_ASTC_12x10_SRGB_BLOCK	$12 \times 10$	Yes	LDR
VK_FORMAT_ASTC_12x12_UNORM_BLOCK	$12 \times 12$	No	LDR
VK_FORMAT_ASTC_12x12_SRGB_BLOCK	$12 \times 12$	Yes	LDR
VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT	$4 \times 4$	No	HDR

<b>VkFormat</b>	<b>Compressed texel block dimensions</b>	<b>sRGB-encoded</b>	<b>Profile</b>
VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT	5 × 4	No	HDR
VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT	5 × 5	No	HDR
VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT	6 × 5	No	HDR
VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT	6 × 6	No	HDR
VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT	8 × 5	No	HDR
VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT	8 × 6	No	HDR
VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT	8 × 8	No	HDR
VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT	10 × 5	No	HDR
VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT	10 × 6	No	HDR
VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT	10 × 8	No	HDR
VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT	10 × 10	No	HDR
VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT	12 × 10	No	HDR
VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT	12 × 12	No	HDR

## ASTC decode mode

If the `VK_EXT_astc_decode_mode` extension is enabled, the decode mode is determined as follows:

*Table 82. Mapping of Vulkan ASTC decoding format to ASTC decoding modes*

<b>VkFormat</b>	<b>Decoding mode</b>
VK_FORMAT_R16G16B16A16_SFLOAT	decode_float16
VK_FORMAT_R8G8B8A8_UNORM	decode_unorm8
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32	decode_rgb9e5

Otherwise, the ASTC decode mode is decode\_float16.

# PVRTC Compressed Image Formats

Table 83. Mapping of Vulkan PVRTC formats to descriptions

VkFormat	Compressed texel block dimensions	sRGB-encoded
VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG	$8 \times 4$	No
VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG	$4 \times 4$	No
VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG	$8 \times 4$	No
VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG	$4 \times 4$	No
VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG	$8 \times 4$	Yes
VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG	$4 \times 4$	Yes
VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG	$8 \times 4$	Yes
VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG	$4 \times 4$	Yes

# Appendix D: Core Revisions (Informative)

New minor versions of the Vulkan API are defined periodically by the Khronos Vulkan Working Group. These consist of some amount of additional functionality added to the core API, potentially including both new functionality and functionality [promoted](#) from extensions.

It is possible to build the specification for earlier versions, but to aid readability of the latest versions, this appendix gives an overview of the changes as compared to earlier versions.

## Version 1.1

Vulkan Version 1.1 [promoted](#) a number of key extensions into the core API:

- [VK\\_KHR\\_16bit\\_storage](#)
- [VK\\_KHR\\_bind\\_memory2](#)
- [VK\\_KHR\\_dedicated\\_allocation](#)
- [VK\\_KHR\\_descriptor\\_update\\_template](#)
- [VK\\_KHR\\_device\\_group](#)
- [VK\\_KHR\\_device\\_group\\_creation](#)
- [VK\\_KHR\\_external\\_fence](#)
- [VK\\_KHR\\_external\\_fence\\_capabilities](#)
- [VK\\_KHR\\_external\\_memory](#)
- [VK\\_KHR\\_external\\_memory\\_capabilities](#)
- [VK\\_KHR\\_external\\_semaphore](#)
- [VK\\_KHR\\_external\\_semaphore\\_capabilities](#)
- [VK\\_KHR\\_get\\_memory\\_requirements2](#)

- [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)
- [VK\\_KHR\\_maintenance1](#)
- [VK\\_KHR\\_maintenance2](#)
- [VK\\_KHR\\_maintenance3](#)
- [VK\\_KHR\\_multiview](#)
- [VK\\_KHR\\_relaxed\\_block\\_layout](#)
- [VK\\_KHR\\_sampler\\_ycbcr\\_conversion](#)
- [VK\\_KHR\\_shader\\_draw\\_parameters](#)
- [VK\\_KHR\\_storage\\_buffer\\_storage\\_class](#)
- [VK\\_KHR\\_variable\\_pointers](#)

The only changes to the functionality added by these extensions were to [VK\\_KHR\\_shader\\_draw\\_parameters](#), which had the `shaderDrawParameters` feature bit added to determine support in the core API, and `variablePointersStorageBuffer` from [VK\\_KHR\\_variable\\_pointers](#) was made optional.

Additionally, Vulkan 1.1 added support for [subgroup](#) operations, [protected memory](#), and a new command to [enumerate the instance version](#).

## New Object Types

- [VkDescriptorUpdateTemplate](#)
- [VkSamplerYcbcrConversion](#)

## New Defines

- [VK\\_API\\_VERSION\\_1\\_1](#)

## New Enum Constants

- Extending [VkBufferCreateFlagBits](#):
  - [VK\\_BUFFER\\_CREATE\\_PROTECTED\\_BIT](#)

- Extending [VkCommandPoolCreateFlagBits](#):
  - `VK_COMMAND_POOL_CREATE_PROTECTED_BIT`
- Extending [VkDependencyFlagBits](#):
  - `VK_DEPENDENCY_DEVICE_GROUP_BIT`
  - `VK_DEPENDENCY_VIEW_LOCAL_BIT`
- Extending [VkDeviceQueueCreateFlagBits](#):
  - `VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT`
- Extending [VkFormat](#):
  - `VK_FORMAT_G8B8G8R8_422_UNORM`
  - `VK_FORMAT_B8G8R8G8_422_UNORM`
  - `VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM`
  - `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM`
  - `VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM`
  - `VK_FORMAT_G8_B8R8_2PLANE_422_UNORM`
  - `VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM`
  - `VK_FORMAT_R10X6_UNORM_PACK16`
  - `VK_FORMAT_R10X6G10X6_UNORM_2PACK16`
  - `VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16`
  - `VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16`
  - `VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16`
  - `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16`
  - `VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16`
  - `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16`
  - `VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16`
  - `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16`
  - `VK_FORMAT_R12X4_UNORM_PACK16`
  - `VK_FORMAT_R12X4G12X4_UNORM_2PACK16`
  - `VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16`
  - `VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16`
  - `VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16`
  - `VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16`
  - `VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16`
  - `VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16`
  - `VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16`
  - `VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16`
  - `VK_FORMAT_G16B16G16R16_422_UNORM`
  - `VK_FORMAT_B16G16R16G16_422_UNORM`
  - `VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM`
  - `VK_FORMAT_G16_B16R16_2PLANE_420_UNORM`
  - `VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM`
  - `VK_FORMAT_G16_B16R16_2PLANE_422_UNORM`

- `VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM`
- Extending `VkFormatFeatureFlagBits`:
  - `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`
  - `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`
  - `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT`
  - `VK_FORMAT_FEATURE_DISJOINT_BIT`
  - `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`
- Extending `VkImageAspectFlagBits`:
  - `VK_IMAGE_ASPECT_PLANE_0_BIT`
  - `VK_IMAGE_ASPECT_PLANE_1_BIT`
  - `VK_IMAGE_ASPECT_PLANE_2_BIT`
- Extending `VkImageCreateFlagBits`:
  - `VK_IMAGE_CREATE_ALIAS_BIT`
  - `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`
  - `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`
  - `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT`
  - `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT`
  - `VK_IMAGE_CREATE_PROTECTED_BIT`
  - `VK_IMAGE_CREATE_DISJOINT_BIT`
- Extending `VkImageLayoutFlagBits`:
  - `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`
  - `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`
- Extending `VkMemoryHeapFlagBits`:
  - `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT`
- Extending `VkMemoryPropertyFlagBits`:
  - `VK_MEMORY_PROPERTY_PROTECTED_BIT`
- Extending `VkObjectType`:
  - `VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION`
  - `VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE`
- Extending `VkPipelineCreateFlagBits`:
  - `VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT`
  - `VK_PIPELINE_CREATE_DISPATCH_BASE`
- Extending `VkQueueFlagBits`:
  - `VK_QUEUE_PROTECTED_BIT`
- Extending `VkResult`:

- VK\_ERROR\_OUT\_OF\_POOL\_MEMORY
  - VK\_ERROR\_INVALID\_EXTERNAL\_HANDLE
- Extending `VkStructureType`:
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_SUBGROUP\_PROPERTIES
    - VK\_STRUCTURE\_TYPE\_BIND\_BUFFER\_MEMORY\_INFO
    - VK\_STRUCTURE\_TYPE\_BIND\_IMAGE\_MEMORY\_INFO
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_16BIT\_STORAGE\_FEATURES
    - VK\_STRUCTURE\_TYPE\_MEMORY\_DEDICATED\_REQUIREMENTS
    - VK\_STRUCTURE\_TYPE\_MEMORY\_DEDICATED\_ALLOCATE\_INFO
    - VK\_STRUCTURE\_TYPE\_MEMORY\_ALLOCATE\_FLAGS\_INFO
    - VK\_STRUCTURE\_TYPE\_DEVICE\_GROUP\_RENDER\_PASS\_BEGIN\_INFO
    - VK\_STRUCTURE\_TYPE\_DEVICE\_GROUP\_COMMAND\_BUFFER\_BEGIN\_INFO
    - VK\_STRUCTURE\_TYPE\_DEVICE\_GROUP\_SUBMIT\_INFO
    - VK\_STRUCTURE\_TYPE\_DEVICE\_GROUP\_BIND\_SPARSE\_INFO
    - VK\_STRUCTURE\_TYPE\_BIND\_BUFFER\_MEMORY\_DEVICE\_GROUP\_INFO
    - VK\_STRUCTURE\_TYPE\_BIND\_IMAGE\_MEMORY\_DEVICE\_GROUP\_INFO
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_GROUP\_PROPERTIES
    - VK\_STRUCTURE\_TYPE\_DEVICE\_GROUP\_DEVICE\_CREATE\_INFO
    - VK\_STRUCTURE\_TYPE\_BUFFER\_MEMORY\_REQUIREMENTS\_INFO\_2
    - VK\_STRUCTURE\_TYPE\_IMAGE\_MEMORY\_REQUIREMENTS\_INFO\_2
    - VK\_STRUCTURE\_TYPE\_IMAGE\_SPARSE\_MEMORY\_REQUIREMENTS\_INFO\_2
    - VK\_STRUCTURE\_TYPE\_MEMORY\_REQUIREMENTS\_2
    - VK\_STRUCTURE\_TYPE\_SPARSE\_IMAGE\_MEMORY\_REQUIREMENTS\_2
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_FEATURES\_2
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_PROPERTIES\_2
    - VK\_STRUCTURE\_TYPE\_FORMAT\_PROPERTIES\_2
    - VK\_STRUCTURE\_TYPE\_IMAGE\_FORMAT\_PROPERTIES\_2
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_IMAGE\_FORMAT\_INFO\_2
    - VK\_STRUCTURE\_TYPE\_QUEUE\_FAMILY\_PROPERTIES\_2
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_MEMORY\_PROPERTIES\_2
    - VK\_STRUCTURE\_TYPE\_SPARSE\_IMAGE\_FORMAT\_PROPERTIES\_2
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_SPARSE\_IMAGE\_FORMAT\_INFO\_2
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_POINT\_CLIPPING\_PROPERTIES
    - VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_INPUT\_ATTACHMENT\_ASPECT\_CREATE\_INFO
    - VK\_STRUCTURE\_TYPE\_IMAGE\_VIEW\_USAGE\_CREATE\_INFO
    - VK\_STRUCTURE\_TYPE\_PIPELINE\_TESSELLATION\_DOMAIN\_ORIGIN\_STATE\_CREATE\_INFO
    - VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_MULTIVIEW\_CREATE\_INFO
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_MULTIVIEW\_FEATURES
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_MULTIVIEW\_PROPERTIES
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_VARIABLE\_POINTERS\_FEATURES
    - VK\_STRUCTURE\_TYPE\_PROTECTED\_SUBMIT\_INFO
    - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_PROTECTED\_MEMORY\_FEATURES

- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_PROTECTED\_MEMORY\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_DEVICE\_QUEUE\_INFO\_2
- VK\_STRUCTURE\_TYPE\_SAMPLER\_YCBCR\_CONVERSION\_CREATE\_INFO
- VK\_STRUCTURE\_TYPE\_SAMPLER\_YCBCR\_CONVERSION\_INFO
- VK\_STRUCTURE\_TYPE\_BIND\_IMAGE\_PLANE\_MEMORY\_INFO
- VK\_STRUCTURE\_TYPE\_IMAGE\_PLANE\_MEMORY\_REQUIREMENTS\_INFO
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_SAMPLER\_YCBCR\_CONVERSION\_FEATURES
- VK\_STRUCTURE\_TYPE\_SAMPLER\_YCBCR\_CONVERSION\_IMAGE\_FORMAT\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_UPDATE\_TEMPLATE\_CREATE\_INFO
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_EXTERNAL\_IMAGE\_FORMAT\_INFO
- VK\_STRUCTURE\_TYPE\_EXTERNAL\_IMAGE\_FORMAT\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_EXTERNAL\_BUFFER\_INFO
- VK\_STRUCTURE\_TYPE\_EXTERNAL\_BUFFER\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_ID\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_EXTERNAL\_MEMORY\_BUFFER\_CREATE\_INFO
- VK\_STRUCTURE\_TYPE\_EXTERNAL\_MEMORY\_IMAGE\_CREATE\_INFO
- VK\_STRUCTURE\_TYPE\_EXPORT\_MEMORY\_ALLOCATE\_INFO
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_EXTERNAL\_FENCE\_INFO
- VK\_STRUCTURE\_TYPE\_EXTERNAL\_FENCE\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_EXPORT\_FENCE\_CREATE\_INFO
- VK\_STRUCTURE\_TYPE\_EXPORT\_SEMAPHORE\_CREATE\_INFO
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_EXTERNAL\_SEMAPHORE\_INFO
- VK\_STRUCTURE\_TYPE\_EXTERNAL\_SEMAPHORE\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_MAINTENANCE\_3\_PROPERTIES
- VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_SET\_LAYOUT\_SUPPORT
- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_SHADER\_DRAW\_PARAMETERS\_FEATURES

## New Enums

- [VkChromaLocation](#)
- [VkDescriptorUpdateTemplateType](#)
- [VkExternalFenceFeatureFlagBits](#)
- [VkExternalFenceHandleTypeFlagBits](#)
- [VkExternalMemoryFeatureFlagBits](#)
- [VkExternalMemoryHandleTypeFlagBits](#)
- [VkExternalSemaphoreFeatureFlagBits](#)
- [VkExternalSemaphoreHandleTypeFlagBits](#)
- [VkFenceImportFlagBits](#)
- [VkMemoryAllocateFlagBits](#)
- [VkPeerMemoryFeatureFlagBits](#)

- [VkPointClippingBehavior](#)
- [VkSamplerYcbcrModelConversion](#)
- [VkSamplerYcbcrRange](#)
- [VkSemaphoreImportFlagBits](#)
- [VkSubgroupFeatureFlagBits](#)
- [VkTessellationDomainOrigin](#)
- [VkCommandPoolTrimFlags](#)
- [VkDescriptorUpdateTemplateCreateFlags](#)
- [VkExternalFenceFeatureFlags](#)
- [VkExternalFenceHandleTypeFlags](#)
- [VkExternalMemoryFeatureFlags](#)
- [VkExternalMemoryHandleTypeFlags](#)
- [VkExternalSemaphoreFeatureFlags](#)
- [VkExternalSemaphoreHandleTypeFlags](#)
- [VkFenceImportFlags](#)
- [VkMemoryAllocateFlags](#)
- [VkPeerMemoryFeatureFlags](#)
- [VkSemaphoreImportFlags](#)
- [VkSubgroupFeatureFlags](#)

## New Structures

- [VkBindBufferMemoryDeviceGroupInfo](#)
- [VkBindBufferMemoryInfo](#)
- [VkBindImageMemoryDeviceGroupInfo](#)
- [VkBindImageMemoryInfo](#)
- [VkBindImagePlaneMemoryInfo](#)
- [VkBufferMemoryRequirementsInfo2](#)
- [VkDescriptorSetLayoutSupport](#)
- [VkDescriptorUpdateTemplateCreateInfo](#)
- [VkDescriptorUpdateTemplateEntry](#)
- [VkDeviceGroupBindSparseInfo](#)
- [VkDeviceGroupCommandBufferBeginInfo](#)
- [VkDeviceGroupDeviceCreateInfo](#)
- [VkDeviceGroupRenderPassBeginInfo](#)
- [VkDeviceGroupSubmitInfo](#)

- [VkDeviceQueueCreateInfo2](#)
- [VkExportFenceCreateInfo](#)
- [VkExportMemoryAllocateCreateInfo](#)
- [VkExportSemaphoreCreateInfo](#)
- [VkExternalBufferProperties](#)
- [VkExternalFenceProperties](#)
- [VkExternalImageFormatProperties](#)
- [VkExternalMemoryBufferCreateInfo](#)
- [VkExternalMemoryImageCreateInfo](#)
- [VkExternalMemoryProperties](#)
- [VkExternalSemaphoreProperties](#)
- [VkFormatProperties2](#)
- [VkImageFormatProperties2](#)
- [VkImageMemoryRequirementsInfo2](#)
- [VkImagePlaneMemoryRequirementsInfo](#)
- [VkImageSparseMemoryRequirementsInfo2](#)
- [VkImageViewUsageCreateInfo](#)
- [VkInputAttachmentAspectReference](#)
- [VkMemoryAllocateFlagsInfo](#)
- [VkMemoryDedicatedAllocateCreateInfo](#)
- [VkMemoryDedicatedRequirements](#)
- [VkMemoryRequirements2](#)
- [VkPhysicalDevice16BitStorageFeatures](#)
- [VkPhysicalDeviceExternalBufferInfo](#)
- [VkPhysicalDeviceExternalFenceInfo](#)
- [VkPhysicalDeviceExternalImageFormatInfo](#)
- [VkPhysicalDeviceExternalSemaphoreInfo](#)
- [VkPhysicalDeviceFeatures2](#)
- [VkPhysicalDeviceGroupProperties](#)
- [VkPhysicalDeviceIDProperties](#)
- [VkPhysicalDeviceImageFormatInfo2](#)
- [VkPhysicalDeviceMaintenance3Properties](#)
- [VkPhysicalDeviceMemoryProperties2](#)
- [VkPhysicalDeviceMultiviewFeatures](#)
- [VkPhysicalDeviceMultiviewProperties](#)

- [VkPhysicalDevicePointClippingProperties](#)
- [VkPhysicalDeviceProperties2](#)
- [VkPhysicalDeviceProtectedMemoryFeatures](#)
- [VkPhysicalDeviceProtectedMemoryProperties](#)
- [VkPhysicalDeviceSamplerYcbcrConversionFeatures](#)
- [VkPhysicalDeviceShaderDrawParametersFeatures](#)
- [VkPhysicalDeviceSparseImageFormatCreateInfo2](#)
- [VkPhysicalDeviceSubgroupProperties](#)
- [VkPhysicalDeviceVariablePointersFeatures](#)
- [VkPipelineTessellationDomainOriginStateCreateInfo](#)
- [VkProtectedSubmitInfo](#)
- [VkQueueFamilyProperties2](#)
- [VkRenderPassInputAttachmentAspectCreateInfo](#)
- [VkRenderPassMultiviewCreateInfo](#)
- [VkSamplerYcbcrConversionCreateInfo](#)
- [VkSamplerYcbcrConversionImageFormatProperties](#)
- [VkSamplerYcbcrConversionCreateInfo](#)
- [VkSparseImageFormatProperties2](#)
- [VkSparseImageMemoryRequirements2](#)

## New Functions

- [vkBindBufferMemory2](#)
- [vkBindImageMemory2](#)
- [vkCmdDispatchBase](#)
- [vkCmdSetDeviceMask](#)
- [vkCreateDescriptorUpdateTemplate](#)
- [vkCreateSamplerYcbcrConversion](#)
- [vkDestroyDescriptorUpdateTemplate](#)
- [vkDestroySamplerYcbcrConversion](#)
- [vkEnumerateInstanceVersion](#)
- [vkEnumeratePhysicalDeviceGroups](#)
- [vkGetBufferMemoryRequirements2](#)
- [vkGetDescriptorSetLayoutSupport](#)
- [vkGetDeviceGroupPeerMemoryFeatures](#)
- [vkGetDeviceQueue2](#)

- [vkGetImageMemoryRequirements2](#)
- [vkGetImageSparseMemoryRequirements2](#)
- [vkGetPhysicalDeviceExternalBufferProperties](#)
- [vkGetPhysicalDeviceExternalFenceProperties](#)
- [vkGetPhysicalDeviceExternalSemaphoreProperties](#)
- [vkGetPhysicalDeviceFeatures2](#)
- [vkGetPhysicalDeviceFormatProperties2](#)
- [vkGetPhysicalDeviceImageFormatProperties2](#)
- [vkGetPhysicalDeviceMemoryProperties2](#)
- [vkGetPhysicalDeviceProperties2](#)
- [vkGetPhysicalDeviceQueueFamilyProperties2](#)
- [vkGetPhysicalDeviceSparseImageFormatProperties2](#)
- [vkTrimCommandPool](#)
- [vkUpdateDescriptorSetWithTemplate](#)

# Appendix E: Layers & Extensions (Informative)

Extensions to the Vulkan API **can** be defined by authors, groups of authors, and the Khronos Vulkan Working Group. In order not to compromise the readability of the Vulkan Specification, the core Specification does not incorporate most extensions. The online Registry of extensions is available at URL

<https://www.khronos.org/registry/vulkan/>

and allows generating versions of the Specification incorporating different extensions.

Most of the content previously in this appendix does not specify **use** of specific Vulkan extensions and layers, but rather specifies the processes by which extensions and layers are created. As of version 1.0.21 of the Vulkan Specification, this content has been migrated to the [Vulkan Documentation and Extensions](#) document. Authors creating extensions and layers **must** follow the mandatory procedures in that document.

The remainder of this appendix documents a set of extensions chosen when this document was built. Versions of the Specification published in the Registry include:

- Core API + mandatory extensions required of all Vulkan implementations.
- Core API + all registered and published Khronos (**KHR**) extensions.
- Core API + all registered and published extensions.

Extensions are grouped as Khronos **KHR**, multivendor **EXT**, and then alphabetically by author ID. Within each group, extensions are listed in alphabetical order by their name.

## Note

As of the initial Vulkan 1.1 public release, the **KHX** author ID is no longer used. All **KHX** extensions have been promoted to **KHR** status. Previously, this author ID was used to indicate that an extension was experimental, and is being considered for standardization in future **KHR** or core Vulkan API versions. We no longer use this mechanism for exposing experimental functionality.



Some vendors may use an alternate author ID ending in **X** for some of their extensions. The exact meaning of such an author ID is defined by each vendor, and may not be equivalent to **KHX**, but it is likely to indicate a lesser degree of interface stability than a non-**X** extension from the same vendor.

## List of Current Extensions

- [VK\\_KHR\\_8bit\\_storage](#)
- [VK\\_KHR\\_android\\_surface](#)
- [VK\\_KHR\\_buffer\\_device\\_address](#)

- [VK\\_KHR\\_create\\_renderpass2](#)
- [VK\\_KHR\\_depth\\_stencil\\_resolve](#)
- [VK\\_KHR\\_display](#)
- [VK\\_KHR\\_display\\_swapchain](#)
- [VK\\_KHR\\_draw\\_indirect\\_count](#)
- [VK\\_KHR\\_driver\\_properties](#)
- [VK\\_KHR\\_external\\_fence\\_fd](#)
- [VK\\_KHR\\_external\\_fence\\_win32](#)
- [VK\\_KHR\\_external\\_memory\\_fd](#)
- [VK\\_KHR\\_external\\_memory\\_win32](#)
- [VK\\_KHR\\_external\\_semaphore\\_fd](#)
- [VK\\_KHR\\_external\\_semaphore\\_win32](#)
- [VK\\_KHR\\_get\\_display\\_properties2](#)
- [VK\\_KHR\\_get\\_surface\\_capabilities2](#)
- [VK\\_KHR\\_image\\_format\\_list](#)
- [VK\\_KHR\\_imageless\\_framebuffer](#)
- [VK\\_KHR\\_incremental\\_present](#)
- [VK\\_KHR\\_performance\\_query](#)
- [VK\\_KHR\\_pipeline\\_executable\\_properties](#)
- [VK\\_KHR\\_push\\_descriptor](#)
- [VK\\_KHR\\_sampler\\_mirror\\_clamp\\_to\\_edge](#)
- [VK\\_KHR\\_separate\\_depth\\_stencil\\_layouts](#)
- [VK\\_KHR\\_shader\\_atomic\\_int64](#)
- [VK\\_KHR\\_shader\\_clock](#)
- [VK\\_KHR\\_shader\\_float16\\_int8](#)
- [VK\\_KHR\\_shader\\_float\\_controls](#)
- [VK\\_KHR\\_shader\\_subgroup\\_extended\\_types](#)
- [VK\\_KHR\\_shared\\_presentable\\_image](#)
- [VK\\_KHR\\_spirv\\_1\\_4](#)
- [VK\\_KHR\\_surface](#)
- [VK\\_KHR\\_surface\\_protected\\_capabilities](#)
- [VK\\_KHR\\_swapchain](#)
- [VK\\_KHR\\_swapchain\\_mutable\\_format](#)
- [VK\\_KHR\\_timeline\\_semaphore](#)
- [VK\\_KHR\\_uniform\\_buffer\\_standard\\_layout](#)

- [VK\\_KHR\\_vulkan\\_memory\\_model](#)
- [VK\\_KHR\\_wayland\\_surface](#)
- [VK\\_KHR\\_win32\\_keyed\\_mutex](#)
- [VK\\_KHR\\_win32\\_surface](#)
- [VK\\_KHR\\_xcb\\_surface](#)
- [VK\\_KHR\\_xlib\\_surface](#)
- [VK\\_EXT\\_acquire\\_xlib\\_display](#)
- [VK\\_EXT\\_astc\\_decode\\_mode](#)
- [VK\\_EXT\\_blend\\_operation\\_advanced](#)
- [VK\\_EXT\\_calibrated\\_timestamps](#)
- [VK\\_EXT\\_conditional\\_rendering](#)
- [VK\\_EXT\\_conservative\\_rasterization](#)
- [VK\\_EXT\\_debug\\_utils](#)
- [VK\\_EXT\\_depth\\_clip\\_enable](#)
- [VK\\_EXT\\_depth\\_range\\_unrestricted](#)
- [VK\\_EXT\\_descriptor\\_indexing](#)
- [VK\\_EXT\\_direct\\_mode\\_display](#)
- [VK\\_EXT\\_discard\\_rectangles](#)
- [VK\\_EXT\\_display\\_control](#)
- [VK\\_EXT\\_display\\_surface\\_counter](#)
- [VK\\_EXT\\_external\\_memory\\_dma\\_buf](#)
- [VK\\_EXT\\_external\\_memory\\_host](#)
- [VK\\_EXT\\_filter\\_cubic](#)
- [VK\\_EXT\\_fragment\\_density\\_map](#)
- [VK\\_EXT\\_fragment\\_shader\\_interlock](#)
- [VK\\_EXT\\_full\\_screen\\_exclusive](#)
- [VK\\_EXT\\_global\\_priority](#)
- [VK\\_EXT\\_hdr\\_metadata](#)
- [VK\\_EXT\\_headless\\_surface](#)
- [VK\\_EXT\\_host\\_query\\_reset](#)
- [VK\\_EXT\\_image\\_drm\\_format\\_modifier](#)
- [VK\\_EXT\\_index\\_type\\_uint8](#)
- [VK\\_EXT\\_inline\\_uniform\\_block](#)
- [VK\\_EXT\\_line\\_rasterization](#)
- [VK\\_EXT\\_memory\\_budget](#)

- [VK\\_EXT\\_memory\\_priority](#)
- [VK\\_EXT\\_metal\\_surface](#)
- [VK\\_EXT\\_pci\\_bus\\_info](#)
- [VK\\_EXT\\_pipeline\\_creation\\_feedback](#)
- [VK\\_EXT\\_post\\_depth\\_coverage](#)
- [VK\\_EXT\\_queue\\_family\\_foreign](#)
- [VK\\_EXT\\_sample\\_locations](#)
- [VK\\_EXT\\_sampler\\_filter\\_minmax](#)
- [VK\\_EXT\\_scalar\\_block\\_layout](#)
- [VK\\_EXT\\_separate\\_stencil\\_usage](#)
- [VK\\_EXT\\_shader\\_demote\\_to\\_helper\\_invocation](#)
- [VK\\_EXT\\_shader\\_stencil\\_export](#)
- [VK\\_EXT\\_shader\\_subgroup\\_ballot](#)
- [VK\\_EXT\\_shader\\_subgroup\\_vote](#)
- [VK\\_EXT\\_shader\\_viewport\\_index\\_layer](#)
- [VK\\_EXT\\_subgroup\\_size\\_control](#)
- [VK\\_EXT\\_swapchain\\_colorspace](#)
- [VK\\_EXT\\_texel\\_buffer\\_alignment](#)
- [VK\\_EXT\\_texture\\_compression\\_astc\\_hdr](#)
- [VK\\_EXT\\_transform\\_feedback](#)
- [VK\\_EXT\\_validation\\_cache](#)
- [VK\\_EXT\\_validation\\_features](#)
- [VK\\_EXT\\_vertex\\_attribute\\_divisor](#)
- [VK\\_EXT\\_ycbcr\\_image\\_arrays](#)
- [VK\\_AMD\\_buffer\\_marker](#)
- [VK\\_AMD\\_device\\_coherent\\_memory](#)
- [VK\\_AMD\\_display\\_native\\_hdr](#)
- [VK\\_AMD\\_gcn\\_shader](#)
- [VK\\_AMD\\_memory\\_overallocation\\_behavior](#)
- [VK\\_AMD\\_mixed\\_attachment\\_samples](#)
- [VK\\_AMD\\_pipeline\\_compiler\\_control](#)
- [VK\\_AMD\\_rasterization\\_order](#)
- [VK\\_AMD\\_shader\\_ballot](#)
- [VK\\_AMD\\_shader\\_core\\_properties](#)
- [VK\\_AMD\\_shader\\_core\\_properties2](#)

- [VK\\_AMD\\_shader\\_explicit\\_vertex\\_parameter](#)
- [VK\\_AMD\\_shader\\_fragment\\_mask](#)
- [VK\\_AMD\\_shader\\_image\\_load\\_store\\_lod](#)
- [VK\\_AMD\\_shader\\_info](#)
- [VK\\_AMD\\_shader\\_trinary\\_minmax](#)
- [VK\\_AMD\\_texture\\_gather\\_bias\\_lod](#)
- [VK\\_ANDROID\\_external\\_memory\\_android\\_hardware\\_buffer](#)
- [VK\\_FUCHSIA\\_imagepipe\\_surface](#)
- [VK\\_GGP\\_frame\\_token](#)
- [VK\\_GGP\\_stream\\_descriptor\\_surface](#)
- [VK\\_GOOGLE\\_decorate\\_string](#)
- [VK\\_GOOGLE\\_display\\_timing](#)
- [VK\\_GOOGLE\\_hlsl\\_functionality1](#)
- [VK\\_GOOGLE\\_user\\_type](#)
- [VK\\_IMG\\_filter\\_cubic](#)
- [VK\\_IMG\\_format\\_pvrtc](#)
- [VK\\_INTEL\\_performance\\_query](#)
- [VK\\_INTEL\\_shader\\_integer\\_functions2](#)
- [VK\\_MVK\\_ios\\_surface](#)
- [VK\\_MVK\\_macos\\_surface](#)
- [VK\\_NN\\_vi\\_surface](#)
- [VK\\_NVX\\_device\\_generated\\_commands](#)
- [VK\\_NVX\\_image\\_view\\_handle](#)
- [VK\\_NVX\\_multiview\\_per\\_view\\_attributes](#)
- [VK\\_NV\\_clip\\_space\\_w\\_scaling](#)
- [VK\\_NV\\_compute\\_shader\\_derivatives](#)
- [VK\\_NV\\_cooperative\\_matrix](#)
- [VK\\_NV\\_corner\\_sampled\\_image](#)
- [VK\\_NV\\_coverage\\_reduction\\_mode](#)
- [VK\\_NV\\_dedicated\\_allocation\\_image\\_aliasing](#)
- [VK\\_NV\\_device\\_diagnostic\\_checkpoints](#)
- [VK\\_NV\\_fill\\_rectangle](#)
- [VK\\_NV\\_fragment\\_coverage\\_to\\_color](#)
- [VK\\_NV\\_fragment\\_shader\\_barycentric](#)
- [VK\\_NV\\_framebuffer\\_mixed\\_samples](#)

- [VK\\_NV\\_geometry\\_shader\\_passthrough](#)
- [VK\\_NV\\_mesh\\_shader](#)
- [VK\\_NV\\_ray\\_tracing](#)
- [VK\\_NV\\_representative\\_fragment\\_test](#)
- [VK\\_NV\\_sample\\_mask\\_override\\_coverage](#)
- [VK\\_NV\\_scissor\\_exclusive](#)
- [VK\\_NV\\_shader\\_image\\_footprint](#)
- [VK\\_NV\\_shader\\_sm\\_builtins](#)
- [VK\\_NV\\_shader\\_subgroup\\_partitioned](#)
- [VK\\_NV\\_shading\\_rate\\_image](#)
- [VK\\_NV\\_viewport\\_array2](#)
- [VK\\_NV\\_viewport\\_swizzle](#)

## VK\_KHR\_8bit\_storage

### Name String

`VK_KHR_8bit_storage`

### Extension Type

Device extension

### Registered Extension Number

178

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_storage_buffer_storage_class`

### Contact

- Alexander Galazin [@legal-arm](#)

### Last Modified Date

2018-02-05

### IP Status

No known IP claims.

### Interactions and External Dependencies

- This extension requires `SPV_KHR_8bit_storage`

### Contributors

- Alexander Galazin, Arm

The `VK_KHR_8bit_storage` extension allows use of 8-bit types in uniform and storage buffers, and push constant blocks. This extension introduces several new optional features which map to SPIR-V capabilities and allow access to 8-bit data in `Block`-decorated objects in the `Uniform` and the `StorageBuffer` storage classes, and objects in the `PushConstant` storage class.

The `StorageBuffer8BitAccess` capability **must** be supported by all implementations of this extension. The other capabilities are optional.

### New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES_KHR`

## New Structures

- [VkPhysicalDevice8BitStorageFeaturesKHR](#)

## New SPIR-V Capabilities

- `StorageBuffer8BitAccess`
- `UniformAndStorageBuffer8BitAccess`
- `StoragePushConstant8`

## Issues

### Version History

- Revision 1, 2018-02-05 (Alexander Galazin)
  - Initial draft

## VK\_KHR\_android\_surface

### Name String

`VK_KHR_android_surface`

### Extension Type

Instance extension

### Registered Extension Number

9

### Revision

6

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### Contact

- Jesse Hall [critsec](#)

### Last Modified Date

2016-01-14

### IP Status

No known IP claims.

### Contributors

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel

- Ian Elliott, LunarG
- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Ray Smith, ARM
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG

The `VK_KHR_android_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to an `ANativeWindow`, Android's native surface type. The `ANativeWindow` represents the producer endpoint of any buffer queue, regardless of consumer endpoint. Common consumer endpoints for `ANativeWindows` are the system window compositor, video encoders, and application-specific compositors importing the images through a `SurfaceTexture`.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_ANDROID_SURFACE_CREATE_INFO_KHR`

## New Enums

None

## New Structures

- `VkAndroidSurfaceCreateInfoKHR`

## New Functions

- `vkCreateAndroidSurfaceKHR`

## Issues

1) Does Android need a way to query for compatibility between a particular physical device (and queue family?) and a specific Android display?

**RESOLVED:** No. Currently on Android, any physical device is expected to be able to present to the system compositor, and all queue families must support the necessary image layout transitions and synchronization operations.

## Version History

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft.
- Revision 2, 2015-10-26 (Ian Elliott)
  - Renamed from VK\_EXT\_KHR\_android\_surface to VK\_KHR\_android\_surface.
- Revision 3, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to surface creation function.
- Revision 4, 2015-11-10 (Jesse Hall)
  - Removed VK\_ERROR\_INVALID\_ANDROID\_WINDOW\_KHR.
- Revision 5, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a pCreateInfo structure.
- Revision 6, 2016-01-14 (James Jones)
  - Moved VK\_ERROR\_NATIVE\_WINDOW\_IN\_USE\_KHR from the VK\_KHR\_android\_surface to the VK\_KHR\_surface extension.

## VK\_KHR\_buffer\_device\_address

### Name String

`VK_KHR_buffer_device_address`

### Extension Type

Device extension

### Registered Extension Number

258

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Jeff Bolz [@jeffbolz](#)

## Last Modified Date

2019-06-24

## IP Status

No known IP claims.

## Contributors

- Jeff Bolz, NVIDIA
- Neil Henning, AMD
- Tobias Hector, AMD
- Jason Ekstrand, Intel
- Baldur Karlsson, Valve
- Jan-Harald Fredriksen, Arm

This extension allows the application to query a 64-bit buffer device address value for a buffer, which can be used to access the buffer memory via the `PhysicalStorageBuffer` storage class in the `GL_EXT_buffer_reference` GLSL extension and `SPV_KHR_physical_storage_buffer` SPIR-V extension.

This extension also allows opaque addresses for buffers and memory objects to be queried and later supplied by a trace capture and replay tool, so that addresses used at replay time match the addresses used when the trace was captured. To enable tools to insert these queries, new memory allocation flags must be specified for memory objects that will be bound to buffers accessed via the `PhysicalStorageBuffer` storage class.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR`
  - `VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO_KHR`
- Extending `VkBufferUsageFlagBits`:
  - `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR`
- Extending `VkBufferCreateFlagBits`:
  - `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`
- Extending `VkMemoryAllocateFlagBits`:
  - `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR`

- `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`
- Extending `VkResult`:
  - `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR`

## New Enums

None

## New Structures

- [VkPhysicalDeviceBufferDeviceAddressFeaturesKHR](#)
- [VkBufferDeviceAddressInfoKHR](#)
- [VkBufferOpaqueCaptureAddressCreateInfoKHR](#)
- [VkMemoryOpaqueCaptureAddressAllocateInfoKHR](#)
- [VkDeviceMemoryOpaqueCaptureAddressCreateInfoKHR](#)

## New Functions

- [vkGetBufferDeviceAddressKHR](#)
- [vkGetBufferOpaqueCaptureAddressKHR](#)
- [vkGetDeviceMemoryOpaqueCaptureAddressKHR](#)

## New Built-In Variables

None

## New SPIR-V Capabilities

- `PhysicalStorageBufferAddresses`

## Issues

None

## Version History

- Revision 1, 2019-06-24 (Jan-Harald Fredriksen)
  - Internal revisions based on VK\_EXT\_buffer\_device\_address

## `VK_KHR_create_renderpass2`

### Name String

`VK_KHR_create_renderpass2`

### Extension Type

Device extension

## Registered Extension Number

110

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK\_KHR\_multiview
- Requires VK\_KHR\_maintenance2

## Contact

- Tobias Hector [@tobias](#)

## Last Modified Date

2018-02-07

## Contributors

- Tobias Hector
- Jeff Bolz

This extension provides a new entry point to create render passes in a way that can be easily extended by other extensions through the substructures of render pass creation. The Vulkan 1.0 render pass creation sub-structures do not include `sType/pNext` members. Additionally, the renderpass begin/next/end commands have been augmented with new extensible structures for passing additional subpass information.

The `VkRenderPassMultiviewCreateInfo` and `VkInputAttachmentAspectReference` structures that extended the original `VkRenderPassCreateInfo` are not accepted into the new creation functions, and instead their parameters are folded into this extension as follows:

- Elements of `VkRenderPassMultiviewCreateInfo::pViewMasks` are now specified in `VkSubpassDescription2KHR::viewMask`.
- Elements of `VkRenderPassMultiviewCreateInfo::pViewOffsets` are now specified in `VkSubpassDependency2KHR::viewOffset`.
- `VkRenderPassMultiviewCreateInfo::correlationMaskCount` and `VkRenderPassMultiviewCreateInfo::pCorrelationMasks` are directly specified in `VkRenderPassCreateInfo2KHR`.
- `VkInputAttachmentAspectReference::aspectMask` is now specified in the relevant input attachment description in `VkAttachmentDescription2KHR::aspectMask`

The details of these mappings are explained fully in the new structures.

## New Enum Constants

- Extending `VkStructureType`:

- `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2_KHR`
- `VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2_KHR`
- `VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2_KHR`
- `VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2_KHR`
- `VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2_KHR`
- `VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO_KHR`
- `VK_STRUCTURE_TYPE_SUBPASS_END_INFO_KHR`

## New Structures

- [VkAttachmentDescription2KHR](#)
- [VkAttachmentReference2KHR](#)
- [VkSubpassDescription2KHR](#)
- [VkSubpassDependency2KHR](#)
- [VkRenderPassCreateInfo2KHR](#)
- [VkSubpassBeginInfoKHR](#)
- [VkSubpassEndInfoKHR](#)

## New Functions

- [vkCreateRenderPass2KHR](#)
- [vkCmdBeginRenderPass2KHR](#)
- [vkCmdNextSubpass2KHR](#)
- [vkCmdEndRenderPass2KHR](#)

## Version History

- Revision 1, 2018-02-07 (Tobias Hector)
  - Internal revisions

## VK\_KHR\_depth\_stencil\_resolve

### Name String

`VK_KHR_depth_stencil_resolve`

### Extension Type

Device extension

### Registered Extension Number

200

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_create\\_renderpass2](#)

## Contact

- Jan-Harald Fredriksen [@janharald](#)

## Last Modified Date

2018-04-09

## Contributors

- Jan-Harald Fredriksen, Arm
- Andrew Garrard, Samsung Electronics
- Soowan Park, Samsung Electronics
- Jeff Bolz, NVIDIA
- Daniel Rakos, AMD

This extension adds support for automatically resolving multisampled depth/stencil attachments in a subpass in a similar manner as for color attachments.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES_KHR`

## New Enums

- [VkResolveModeFlagBitsKHR](#)

## New Structures

- [VkPhysicalDeviceDepthStencilResolvePropertiesKHR](#)
- [VkSubpassDescriptionDepthStencilResolveKHR](#)

## New Functions

None.

## Version History

- Revision 1, 2018-04-09 (Jan-Harald Fredriksen)
  - Initial revision

## VK\_KHR\_display

### Name String

`VK_KHR_display`

### Extension Type

Instance extension

### Registered Extension Number

3

### Revision

23

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### Contact

- James Jones [@cubanismo](#)
- Norbert Nopper [@FslNopper](#)

### Last Modified Date

2017-03-13

### IP Status

No known IP claims.

### Contributors

- James Jones, NVIDIA
- Norbert Nopper, Freescale
- Jeff Vigil, Qualcomm
- Daniel Rakos, AMD

This extension provides the API to enumerate displays and available modes on a given device.

## New Object Types

- [VkDisplayKHR](#)
- [VkDisplayModeKHR](#)

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR`

## New Enums

- [VkDisplayPlaneAlphaFlagBitsKHR](#)

## New Structures

- [VkDisplayPropertiesKHR](#)
- [VkDisplayModeParametersKHR](#)
- [VkDisplayModePropertiesKHR](#)
- [VkDisplayModeCreateInfoKHR](#)
- [VkDisplayPlanePropertiesKHR](#)
- [VkDisplayPlaneCapabilitiesKHR](#)
- [VkDisplaySurfaceCreateInfoKHR](#)

## New Functions

- [vkGetPhysicalDeviceDisplayPropertiesKHR](#)
- [vkGetPhysicalDeviceDisplayPlanePropertiesKHR](#)
- [vkGetDisplayPlaneSupportedDisplaysKHR](#)
- [vkGetDisplayModePropertiesKHR](#)
- [vkCreateDisplayModeKHR](#)
- [vkGetDisplayPlaneCapabilitiesKHR](#)
- [vkCreateDisplayPlaneSurfaceKHR](#)

## Issues

1) Which properties of a mode should be fixed in the mode info vs. settable in some other function when setting the mode? E.g., do we need to double the size of the mode pool to include both stereo and non-stereo modes? YUV and RGB scanout even if they both take RGB input images? BGR vs. RGB input? etc.

**PROPOSED RESOLUTION:** Many modern displays support at most a handful of resolutions and timings natively. Other “modes” are expected to be supported using scaling hardware on the display engine or GPU. Other properties, such as rotation and mirroring should not require duplicating hardware modes just to express all combinations. Further, these properties may be implemented on a per-display or per-overlay granularity.

To avoid the exponential growth of modes as mutable properties are added, as was the case with [EGLConfig](#)/WGL pixel formats/[GLXFBConfig](#), this specification should separate out hardware properties and configurable state into separate objects. Modes and overlay planes will express capabilities of the hardware, while a separate structure will allow applications to configure scaling, rotation, mirroring, color keys, LUT values, alpha masks, etc. for a given swapchain independent of the mode in use. Constraints on these settings will be established by properties of the immutable objects.

Note the resolution of this issue may affect issue 5 as well.

2) What properties of a display itself are useful?

**PROPOSED RESOLUTION:** This issue is too broad. It was meant to prompt general discussion, but resolving this issue amounts to completing this specification. All interesting properties should be included. The issue will remain as a placeholder since removing it would make it hard to parse existing discussion notes that refer to issues by number.

3) How are multiple overlay planes within a display or mode enumerated?

**PROPOSED RESOLUTION:** They are referred to by an index. Each display will report the number of overlay planes it contains.

4) Should swapchains be created relative to a mode or a display?

**PROPOSED RESOLUTION:** When using this extension, swapchains are created relative to a mode and a plane. The mode implies the display object the swapchain will present to. If the specified mode is not the display's current mode, the new mode will be applied when the first image is presented to the swapchain, and the default operating system mode, if any, will be restored when the swapchain is destroyed.

5) Should users query generic ranges from displays and construct their own modes explicitly using those constraints rather than querying a fixed set of modes (Most monitors only have one real "mode" these days, even though many support relatively arbitrary scaling, either on the monitor side or in the GPU display engine, making "modes" something of a relic/compatibility construct).

**PROPOSED RESOLUTION:** Expose both. Display info structures will expose a set of predefined modes, as well as any attributes necessary to construct a customized mode.

6) Is it fine if we return the display and display mode handles in the structure used to query their properties?

**PROPOSED RESOLUTION:** Yes.

7) Is there a possibility that not all displays of a device work with all of the present queues of a device? If yes, how do we determine which displays work with which present queues?

**PROPOSED RESOLUTION:** No known hardware has such limitations, but determining such limitations is supported automatically using the existing `VK_KHR_surface` and `VK_KHR_swapchain` query mechanisms.

8) Should all presentation need to be done relative to an overlay plane, or can a display mode + display be used alone to target an output?

**PROPOSED RESOLUTION:** Require specifying a plane explicitly.

9) Should displays have an associated window system display, such as an `HDC` or `Display`\*?

**PROPOSED RESOLUTION:** No. Displays are independent of any windowing system in use on the system. Further, neither `HDC` nor `Display`\* refer to a physical display object.

10) Are displays queried from a physical GPU or from a device instance?

**PROPOSED RESOLUTION:** Developers prefer to query modes directly from the physical GPU so they can use display information as an input to their device selection algorithms prior to device creation. This avoids the need to create dummy device instances to enumerate displays.

This preference must be weighed against the extra initialization that must be done by driver vendors prior to device instance creation to support this usage.

11) Should displays and/or modes be dispatchable objects? If functions are to take displays, overlays, or modes as their first parameter, they must be dispatchable objects as defined in Khronos bug 13529. If they are not added to the list of dispatchable objects, functions operating on them must take some higher-level object as their first parameter. There is no performance case against making them dispatchable objects, but they would be the first extension objects to be dispatchable.

**PROPOSED RESOLUTION:** Do not make displays or modes dispatchable. They will dispatch based on their associated physical device.

12) Should hardware cursor capabilities be exposed?

**PROPOSED RESOLUTION:** Defer. This could be a separate extension on top of the base WSI specs.

if they are one physical display device to an end user, but may internally be implemented as two side-by-side displays using the same display engine (and sometimes cabling) resources as two physically separate display devices.

**RESOLVED:** Tiled displays will appear as a single display object in this API.

14) Should the raw EDID data be included in the display information?

**RESOLVED:** No. A future extension could be added which reports the EDID if necessary. This may be complicated by the outcome of issue 13.

15) Should min and max scaling factor capabilities of overlays be exposed?

**RESOLVED:** Yes. This is exposed indirectly by allowing applications to query the min/max position and extent of the source and destination regions from which image contents are fetched by the display engine when using a particular mode and overlay pair.

16) Should devices be able to expose planes that can be moved between displays? If so, how?

**RESOLVED:** Yes. Applications can determine which displays a given plane supports using [vkGetDisplayPlaneSupportedDisplaysKHR](#).

17) Should there be a way to destroy display modes? If so, does it support destroying “built in” modes?

**RESOLVED:** Not in this extension. A future extension could add this functionality.

18) What should the lifetime of display and built-in display mode objects be?

**RESOLVED:** The lifetime of the instance. These objects cannot be destroyed. A future extension may be added to expose a way to destroy these objects and/or support display hotplug.

19) Should persistent mode for smart panels be enabled/disabled at swapchain creation time, or on a per-present basis.

**RESOLVED:** On a per-present basis.

## Examples

### Note



The example code for the `VK_KHR_display` and `VK_KHR_display_swapchain` extensions was removed from the appendix after revision 1.0.43. The display enumeration example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: <https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c>).

## Version History

- Revision 1, 2015-02-24 (James Jones)
  - Initial draft
- Revision 2, 2015-03-12 (Norbert Nopper)
  - Added overlay enumeration for a display.
- Revision 3, 2015-03-17 (Norbert Nopper)
  - Fixed typos and namings as discussed in Bugzilla.
  - Reordered and grouped functions.
  - Added functions to query count of display, mode and overlay.
  - Added native display handle, which is maybe needed on some platforms to create a native Window.
- Revision 4, 2015-03-18 (Norbert Nopper)
  - Removed primary and virtualPosition members (see comment of James Jones in Bugzilla).
  - Added native overlay handle to info structure.
  - Replaced , with ; in struct.
- Revision 6, 2015-03-18 (Daniel Rakos)
  - Added WSI extension suffix to all items.
  - Made the whole API more "Vulkanish".
  - Replaced all functions with a single `vkGetDisplayInfoKHR` function to better match the rest of the API.
  - Made the display, display mode, and overlay objects be first class objects, not subclasses of `VkBaseObject` as they do not support the common functions anyways.
  - Renamed \*Info structures to \*Properties.

- Removed overlayIndex field from VkOverlayProperties as there is an implicit index already as a result of moving to a "Vulkanish" API.
  - Displays are not get through device, but through physical GPU to match the rest of the Vulkan API. Also this is something ISVs explicitly requested.
  - Added issue (6) and (7).
- Revision 7, 2015-03-25 (James Jones)
    - Added an issues section
    - Added rotation and mirroring flags
  - Revision 8, 2015-03-25 (James Jones)
    - Combined the duplicate issues sections introduced in last change.
    - Added proposed resolutions to several issues.
  - Revision 9, 2015-04-01 (Daniel Rakos)
    - Rebased extension against Vulkan 0.82.0
  - Revision 10, 2015-04-01 (James Jones)
    - Added issues (10) and (11).
    - Added more straw-man issue resolutions, and cleaned up the proposed resolution for issue (4).
    - Updated the rotation and mirroring enums to have proper bitmask semantics.
  - Revision 11, 2015-04-15 (James Jones)
    - Added proposed resolution for issues (1) and (2).
    - Added issues (12), (13), (14), and (15)
    - Removed pNativeHandle field from overlay structure.
    - Fixed small compilation errors in example code.
  - Revision 12, 2015-07-29 (James Jones)
    - Rewrote the guts of the extension against the latest WSI swapchain specifications and the latest Vulkan API.
    - Address overlay planes by their index rather than an object handle and refer to them as "planes" rather than "overlays" to make it slightly clearer that even a display with no "overlays" still has at least one base "plane" that images can be displayed on.
    - Updated most of the issues.
    - Added an "extension type" section to the specification header.
    - Re-used the VK\_EXT\_KHR\_surface surface transform enumerations rather than redefining them here.
    - Updated the example code to use the new semantics.
  - Revision 13, 2015-08-21 (Ian Elliott)
    - Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.

- Switched from "revision" to "version", including use of the VK\_MAKE\_VERSION macro in the header file.
- Revision 14, 2015-09-01 (James Jones)
  - Restore single-field revision number.
- Revision 15, 2015-09-08 (James Jones)
  - Added alpha flags enum.
  - Added premultiplied alpha support.
- Revision 16, 2015-09-08 (James Jones)
  - Added description section to the spec.
  - Added issues 16 - 18.
- Revision 17, 2015-10-02 (James Jones)
  - Planes are now a property of the entire device rather than individual displays. This allows planes to be moved between multiple displays on devices that support it.
  - Added a function to create a VkSurfaceKHR object describing a display plane and mode to align with the new per-platform surface creation conventions.
  - Removed detailed mode timing data. It was agreed that the mode extents and refresh rate are sufficient for current use cases. Other information could be added back<sup>2</sup> in as an extension if it is needed in the future.
  - Added support for smart/persistent/buffered display devices.
- Revision 18, 2015-10-26 (Ian Elliott)
  - Renamed from VK\_EXT\_KHR\_display to VK\_KHR\_display.
- Revision 19, 2015-11-02 (James Jones)
  - Updated example code to match revision 17 changes.
- Revision 20, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to creation functions.
- Revision 21, 2015-11-10 (Jesse Hall)
  - Added VK\_DISPLAY\_PLANE\_ALPHA\_OPAQUE\_BIT\_KHR, and use VkDisplayPlaneAlphaFlagBitsKHR for VkDisplayPlanePropertiesKHR::alphaMode instead of VkDisplayPlaneAlphaFlagsKHR, since it only represents one mode.
  - Added reserved flags bitmask to VkDisplayPlanePropertiesKHR.
  - Use VkSurfaceTransformFlagBitsKHR instead of obsolete VkSurfaceTransformKHR.
  - Renamed vkGetDisplayPlaneSupportedDisplaysKHR parameters for clarity.
- Revision 22, 2015-12-18 (James Jones)
  - Added missing "planeIndex" parameter to vkGetDisplayPlaneSupportedDisplaysKHR()
- Revision 23, 2017-03-13 (James Jones)
  - Closed all remaining issues. The specification and implementations have been shipping with the proposed resolutions for some time now.

- Removed the sample code and noted it has been integrated into the official Vulkan SDK cube demo.

## VK\_KHR\_display\_swapchain

### Name String

`VK_KHR_display_swapchain`

### Extension Type

Device extension

### Registered Extension Number

4

### Revision

10

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_swapchain](#)
- Requires [VK\\_KHR\\_display](#)

### Contact

- James Jones [@cubanismo](#)

### Last Modified Date

2017-03-13

### IP Status

No known IP claims.

### Contributors

- James Jones, NVIDIA
- Jeff Vigil, Qualcomm
- Jesse Hall, Google

This extension provides an API to create a swapchain directly on a device's display without any underlying window system.

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR`

- Extending [VkResult](#):
  - `VK_ERROR_INCOMPATIBLE_DISPLAY_KHR`

## New Enums

None

## New Structures

- [VkDisplayPresentInfoKHR](#)

## New Functions

- [vkCreateSharedSwapchainsKHR](#)

## Issues

1) Should swapchains sharing images each hold a reference to the images, or should it be up to the application to destroy the swapchains and images in an order that avoids the need for reference counting?

**RESOLVED:** Take a reference. The lifetime of presentable images is already complex enough.

2) Should the `srcRect/dstRect` parameters be specified as part of the present command, or at swapchain creation time?

**RESOLVED:** As part of the presentation command. This allows moving and scaling the image on the screen without the need to respecify the mode or create a new swapchain and presentable images.

3) Should `srcRect/dstRect` be specified as rects, or separate offset/extents values?

**RESOLVED:** As rects. Specifying them separately might make it easier for hardware to expose support for one but not the other, but in such cases applications must just take care to obey the reported capabilities and not use non-zero offsets or extents that require scaling, as appropriate.

4) How can applications create multiple swapchains that use the same images?

**RESOLVED:** By calling [vkCreateSharedSwapchainsKHR](#).

An earlier resolution used [vkCreateSwapchainKHR](#), chaining multiple [VkSwapchainCreateInfoKHR](#) structures through `pNext`. In order to allow each swapchain to also allow other extension structs, a level of indirection was used: `VkSwapchainCreateInfoKHR::pNext` pointed to a different structure, which had both an `sType/pNext` for additional extensions, and also had a pointer to the next [VkSwapchainCreateInfoKHR](#) structure. The number of swapchains to be created could only be found by walking this linked list of alternating structures, and the `pSwapchains` out parameter was reinterpreted to be an array of [VkSwapchainKHR](#) handles.

Another option considered was a method to specify a “shared” swapchain when creating a new swapchain, such that groups of swapchains using the same images could be built up one at a time. This was deemed unusable because drivers need to know all of the displays an image will be used

on when determining which internal formats and layouts to use for that image.

## Examples

### Note

The example code for the `VK_KHR_display` and `VK_KHR_display_swapchain` extensions was removed from the appendix after revision 1.0.43. The display swapchain creation example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: <https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c>).

## Version History

- Revision 1, 2015-07-29 (James Jones)
  - Initial draft
- Revision 2, 2015-08-21 (Ian Elliott)
  - Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  - Switched from "revision" to "version", including use of the `VK_MAKE_VERSION` macro in the header file.
- Revision 3, 2015-09-01 (James Jones)
  - Restore single-field revision number.
- Revision 4, 2015-09-08 (James Jones)
  - Allow creating multiple swap chains that share the same images using a single call to `vkCreateSwapChainKHR()`.
- Revision 5, 2015-09-10 (Alon Or-bach)
  - Removed underscores from `SWAP_CHAIN` in two enums.
- Revision 6, 2015-10-02 (James Jones)
  - Added support for smart panels/buffered displays.
- Revision 7, 2015-10-26 (Ian Elliott)
  - Renamed from `VK_EXT_KHR_display_swapchain` to `VK_KHR_display_swapchain`.
- Revision 8, 2015-11-03 (Daniel Rakos)
  - Updated sample code based on the changes to `VK_KHR_swapchain`.
- Revision 9, 2015-11-10 (Jesse Hall)
  - Replaced `VkDisplaySwapchainCreateInfoKHR` with `vkCreateSharedSwapchainsKHR`, changing resolution of issue #4.
- Revision 10, 2017-03-13 (James Jones)
  - Closed all remaining issues. The specification and implementations have been shipping with the proposed resolutions for some time now.

- Removed the sample code and noted it has been integrated into the official Vulkan SDK cube demo.

## **VK\_KHR\_draw\_indirect\_count**

### **Name String**

`VK_KHR_draw_indirect_count`

### **Extension Type**

Device extension

### **Registered Extension Number**

170

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Contact**

- Piers Daniell [Opdaniell-nv](#)

### **Status**

Draft

### **Last Modified Date**

2017-08-25

### **IP Status**

No known IP claims.

### **Contributors**

- Matthaeus G. Chajdas, AMD
- Derrick Owens, AMD
- Graham Sellers, AMD
- Daniel Rakos, AMD
- Dominik Witczak, AMD
- Piers Daniell, NVIDIA

This extension is based off the VK\_AMD\_draw\_indirect\_count extension. This extension allows an application to source the number of draw calls for indirect draw calls from a buffer. This enables applications to generate arbitrary amounts of draw commands and execute them without host intervention.

## New Functions

- [vkCmdDrawIndirectCountKHR](#)
- [vkCmdDrawIndexedIndirectCountKHR](#)

## Version History

- Revision 1, 2017-08-25 (Piers Daniell)
  - Initial draft based off VK\_AMD\_draw\_indirect\_count

## VK\_KHR\_driver\_properties

### Name String

`VK_KHR_driver_properties`

### Extension Type

Device extension

### Registered Extension Number

197

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Daniel Rakos [@drakos-amd](#)

### Last Modified Date

2018-04-11

### IP Status

No known IP claims.

### Contributors

- Baldur Karlsson
- Matthaeus G. Chajdas, AMD
- Piers Daniell, NVIDIA
- Alexander Galazin, Arm
- Jesse Hall, Google
- Daniel Rakos, AMD

This extension provides a new physical device query which allows retrieving information about the

driver implementation, allowing applications to determine which physical device corresponds to which particular vendor's driver, and which conformance test suite version the driver implementation is compliant with.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES_KHR`
- `VK_MAX_DRIVER_NAME_SIZE_KHR`
- `VK_MAX_DRIVER_INFO_SIZE_KHR`

## New Enums

None.

## New Structures

- [VkConformanceVersionKHR](#)
- [VkPhysicalDeviceDriverPropertiesKHR](#)

## New Functions

None.

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2018-04-11 (Daniel Rakos)
  - Internal revisions

## **VK\_KHR\_external\_fence\_fd**

### Name String

`VK_KHR_external_fence_fd`

### Extension Type

Device extension

## Registered Extension Number

116

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_fence](#)

## Contact

- Jesse Hall [@critsec](#)

## Last Modified Date

2017-05-08

## IP Status

No known IP claims.

## Contributors

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Cass Everitt, Oculus
- Contributors to [VK\\_KHR\\_external\\_semaphore\\_fd](#)

An application using external memory may wish to synchronize access to that memory using fences. This extension enables an application to export fence payload to and import fence payload from POSIX file descriptors.

## New Object Types

None.

## New Enum Constants

- [VK\\_STRUCTURE\\_TYPE\\_IMPORT\\_FENCE\\_FD\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_FENCE\\_GET\\_FD\\_INFO\\_KHR](#)

## New Enums

None.

## New Structs

- [VkImportFenceFdInfoKHR](#)

- [VkFenceGetFdInfoKHR](#)

## New Functions

- [vkImportFenceFdKHR](#)
- [vkGetFenceFdKHR](#)

## Issues

This extension borrows concepts, semantics, and language from [VK\\_KHR\\_external\\_semaphore\\_fd](#). That extension's issues apply equally to this extension.

## Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial revision

## **VK\_KHR\_external\_fence\_win32**

### Name String

`VK_KHR_external_fence_win32`

### Extension Type

Device extension

### Registered Extension Number

115

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_fence](#)

### Contact

- Jesse Hall [@critsec](#)

### Last Modified Date

2017-05-08

### IP Status

No known IP claims.

### Contributors

- Jesse Hall, Google
- James Jones, NVIDIA

- Jeff Juliano, NVIDIA
- Cass Everitt, Oculus
- Contributors to [VK\\_KHR\\_external\\_semaphore\\_win32](#)

An application using external memory may wish to synchronize access to that memory using fences. This extension enables an application to export fence payload to and import fence payload from Windows handles.

## New Object Types

None.

## New Enum Constants

- [VK\\_STRUCTURE\\_TYPE\\_IMPORT\\_FENCE\\_WIN32\\_HANDLE\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_EXPORT\\_FENCE\\_WIN32\\_HANDLE\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_FENCE\\_GET\\_WIN32\\_HANDLE\\_INFO\\_KHR](#)

## New Enums

None.

## New Structs

- [VkImportFenceWin32HandleInfoKHR](#)
- [VkExportFenceWin32HandleInfoKHR](#)
- [VkFenceGetWin32HandleInfoKHR](#)

## New Functions

- [vkImportFenceWin32HandleKHR](#)
- [vkGetFenceWin32HandleKHR](#)

## Issues

This extension borrows concepts, semantics, and language from [VK\\_KHR\\_external\\_semaphore\\_win32](#). That extension's issues apply equally to this extension.

1) Should D3D12 fence handle types be supported, like they are for semaphores?

**RESOLVED:** No. Doing so would require extending the fence signal and wait operations to provide values to signal / wait for, like [VkD3D12FenceSubmitInfoKHR](#) does. A D3D12 fence can be signaled by importing it into a [VkSemaphore](#) instead of a [VkFence](#), and applications can check status or wait on the D3D12 fence using non-Vulkan APIs. The convenience of being able to do these operations on [VkFence](#) objects doesn't justify the extra API complexity.

## Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial revision

## VK\_KHR\_external\_memory\_fd

### Name String

`VK_KHR_external_memory_fd`

### Extension Type

Device extension

### Registered Extension Number

75

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_memory](#)

### Contact

- James Jones [@cubanismo](#)

### Last Modified Date

2016-10-21

### IP Status

No known IP claims.

### Contributors

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA

An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension enables an application to export POSIX file descriptor handles from Vulkan memory objects and to import Vulkan memory objects from POSIX file descriptor handles exported from other Vulkan memory objects or from similar resources in other APIs.

## New Object Types

None.

## New Enum Constants

- [VK\\_STRUCTURE\\_TYPE\\_IMPORT\\_MEMORY\\_FD\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_MEMORY\\_FD\\_PROPERTIES\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_MEMORY\\_GET\\_FD\\_INFO\\_KHR](#)

## New Enums

None.

## New Structs

- [VkImportMemoryFdInfoKHR](#)
- [VkMemoryFdPropertiesKHR](#)
- [VkMemoryGetFdInfoKHR](#)

## New Functions

- [vkGetMemoryFdKHR](#)
- [vkGetMemoryFdPropertiesKHR](#)

## Issues

1) Does the application need to close the file descriptor returned by [vkGetMemoryFdKHR](#)?

**RESOLVED:** Yes, unless it is passed back in to a driver instance to import the memory. A successful get call transfers ownership of the file descriptor to the application, and a successful import transfers it back to the driver. Destroying the original memory object will not close the file descriptor or remove its reference to the underlying memory resource associated with it.

2) Do drivers ever need to expose multiple file descriptors per memory object?

**RESOLVED:** No. This would indicate there are actually multiple memory objects, rather than a single memory object.

3) How should the valid size and memory type for POSIX file descriptor memory handles created outside of Vulkan be specified?

**RESOLVED:** The valid memory types are queried directly from the external handle. The size will be specified by future extensions that introduce such external memory handle types.

## Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

## **VK\_KHR\_external\_memory\_win32**

### **Name String**

`VK_KHR_external_memory_win32`

### **Extension Type**

Device extension

### **Registered Extension Number**

74

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_external_memory`

### **Contact**

- James Jones [@cubanismo](#)

### **Last Modified Date**

2016-10-21

### **IP Status**

No known IP claims.

### **Contributors**

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension enables an application to export Windows handles from Vulkan memory objects and to import Vulkan memory objects from Windows handles exported from other Vulkan memory objects or from similar resources in other APIs.

## **New Object Types**

None.

## **New Enum Constants**

- `VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHR`
- `VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR`
- `VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHR`

- [VK\\_STRUCTURE\\_TYPE\\_MEMORY\\_GET\\_WIN32\\_HANDLE\\_INFO\\_KHR](#)

## New Enums

None.

## New Structs

- [VkImportMemoryWin32HandleInfoKHR](#)
- [VkExportMemoryWin32HandleInfoKHR](#)
- [VkMemoryWin32HandlePropertiesKHR](#)
- [VkMemoryGetWin32HandleInfoKHR](#)

## New Functions

- [vkGetMemoryWin32HandleKHR](#)
- [vkGetMemoryWin32HandlePropertiesKHR](#)

## Issues

1) Do applications need to call [CloseHandle\(\)](#) on the values returned from [vkGetMemoryWin32HandleKHR](#) when [handleType](#) is [VK\\_EXTERNAL\\_MEMORY\\_HANDLE\\_TYPE\\_OPAQUE\\_WIN32\\_BIT\\_KHR](#)?

**RESOLVED:** Yes, unless it is passed back in to another driver instance to import the object. A successful get call transfers ownership of the handle to the application. Destroying the memory object will not destroy the handle or the handle's reference to the underlying memory resource.

2) Should the language regarding KMT/Windows 7 handles be moved to a separate extension so that it can be deprecated over time?

**RESOLVED:** No. Support for them can be deprecated by drivers if they choose, by no longer returning them in the supported handle types of the instance level queries.

3) How should the valid size and memory type for windows memory handles created outside of Vulkan be specified?

**RESOLVED:** The valid memory types are queried directly from the external handle. The size is determined by the associated image or buffer memory requirements for external handle types that require dedicated allocations, and by the size specified when creating the object from which the handle was exported for other external handle types.

## Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

## **VK\_KHR\_external\_semaphore\_fd**

### **Name String**

`VK_KHR_external_semaphore_fd`

### **Extension Type**

Device extension

### **Registered Extension Number**

80

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_semaphore](#)

### **Contact**

- James Jones [@cubanismo](#)

### **Last Modified Date**

2016-10-21

### **IP Status**

No known IP claims.

### **Contributors**

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore payload to and import semaphore payload from POSIX file descriptors.

## **New Object Types**

None.

## **New Enum Constants**

- `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHR`
- `VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR`

## New Enums

None.

## New Structs

- [VkImportSemaphoreFdInfoKHR](#)
- [VkSemaphoreGetFdInfoKHR](#)

## New Functions

- [vkImportSemaphoreFdKHR](#)
- [vkGetSemaphoreFdKHR](#)

## Issues

1) Does the application need to close the file descriptor returned by [vkGetSemaphoreFdKHR](#)?

**RESOLVED:** Yes, unless it is passed back in to a driver instance to import the semaphore. A successful get call transfers ownership of the file descriptor to the application, and a successful import transfers it back to the driver. Destroying the original semaphore object will not close the file descriptor or remove its reference to the underlying semaphore resource associated with it.

## Version History

- Revision 1, 2016-10-21 (Jesse Hall)
  - Initial revision

## VK\_KHR\_external\_semaphore\_win32

### Name String

`VK_KHR_external_semaphore_win32`

### Extension Type

Device extension

### Registered Extension Number

79

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_semaphore](#)

### Contact

- James Jones [Qcubanismo](#)

## Last Modified Date

2016-10-21

## IP Status

No known IP claims.

## Contributors

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore payload to and import semaphore payload from Windows handles.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR`
- `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR`
- `VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHR`
- `VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_KHR`

## New Enums

None.

## New Structs

- [VkImportSemaphoreWin32HandleInfoKHR](#)
- [VkExportSemaphoreWin32HandleInfoKHR](#)
- [VkD3D12FenceSubmitInfoKHR](#)
- [VkSemaphoreGetWin32HandleInfoKHR](#)

## New Functions

- [vkImportSemaphoreWin32HandleKHR](#)
- [vkGetSemaphoreWin32HandleKHR](#)

## Issues

1) Do applications need to call `CloseHandle()` on the values returned from `vkGetSemaphoreWin32HandleKHR` when `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR`?

**RESOLVED:** Yes, unless it is passed back in to another driver instance to import the object. A successful get call transfers ownership of the handle to the application. Destroying the semaphore object will not destroy the handle or the handle's reference to the underlying semaphore resource.

2) Should the language regarding KMT/Windows 7 handles be moved to a separate extension so that it can be deprecated over time?

**RESOLVED:** No. Support for them can be deprecated by drivers if they choose, by no longer returning them in the supported handle types of the instance level queries.

3) Should applications be allowed to specify additional object attributes for shared handles?

**RESOLVED:** Yes. Applications will be allowed to provide similar attributes to those they would to any other handle creation API.

4) How do applications communicate the desired fence values to use with `D3D12_FENCE`-based Vulkan semaphores?

**RESOLVED:** There are a couple of options. The values for the signaled and reset states could be communicated up front when creating the object and remain static for the life of the Vulkan semaphore, or they could be specified using auxiliary structures when submitting semaphore signal and wait operations, similar to what is done with the keyed mutex extensions. The latter is more flexible and consistent with the keyed mutex usage, but the former is a much simpler API.

Since Vulkan tends to favor flexibility and consistency over simplicity, a new structure specifying D3D12 fence acquire and release values is added to the `vkQueueSubmit` function.

## Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

## `VK_KHR_get_display_properties2`

### Name String

`VK_KHR_get_display_properties2`

### Extension Type

Instance extension

### Registered Extension Number

122

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_display](#)

## Contact

- James Jones [@cubanismo](#)

## Last Modified Date

2017-02-21

## IP Status

No known IP claims.

## Contributors

- Ian Elliott, Google
- James Jones, NVIDIA

This extension provides new entry points to query device display properties and capabilities in a way that can be easily extended by other extensions, without introducing any further entry points. This extension can be considered the [VK\\_KHR\\_display](#) equivalent of the [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#) extension.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_DISPLAY\\_PROPERTIES\\_2\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_DISPLAY\\_PLANE\\_PROPERTIES\\_2\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_DISPLAY\\_MODE\\_PROPERTIES\\_2\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_DISPLAY\\_PLANE\\_INFO\\_2\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_DISPLAY\\_PLANE\\_CAPABILITIES\\_2\\_KHR](#)

## New Enums

None.

## New Structures

- [VkDisplayProperties2KHR](#)
- [VkDisplayPlaneProperties2KHR](#)

- [VkDisplayModeProperties2KHR](#)
- [VkDisplayPlaneInfo2KHR](#)
- [VkDisplayPlaneCapabilities2KHR](#)

## New Functions

- [vkGetPhysicalDeviceDisplayProperties2KHR](#)
- [vkGetPhysicalDeviceDisplayPlaneProperties2KHR](#)
- [vkGetDisplayModeProperties2KHR](#)
- [vkGetDisplayPlaneCapabilities2KHR](#)

## Issues

1) What should this extension be named?

**RESOLVED:** `VK_KHR_get_display_properties2`. Other alternatives:

- `VK_KHR_display2`
- One extension, combined with `VK_KHR_surface_capabilities2`.

2) Should extensible input structs be added for these new functions:

**RESOLVED:**

- [vkGetPhysicalDeviceDisplayProperties2KHR](#): No. The only current input is a [VkPhysicalDevice](#). Other inputs wouldn't make sense.
- [vkGetPhysicalDeviceDisplayPlaneProperties2KHR](#): No. The only current input is a [VkPhysicalDevice](#). Other inputs wouldn't make sense.
- [vkGetDisplayModeProperties2KHR](#): No. The only current inputs are a [VkPhysicalDevice](#) and a [VkDisplayModeKHR](#). Other inputs wouldn't make sense.

3) Should additional display query functions be extended?

**RESOLVED:**

- [vkGetDisplayPlaneSupportedDisplaysKHR](#): No. Extensions should instead extend [vkGetDisplayPlaneCapabilitiesKHR\(\)](#).

## Version History

- Revision 1, 2017-02-21 (James Jones)
  - Initial draft.

## `VK_KHR_get_surface_capabilities2`

### Name String

`VK_KHR_get_surface_capabilities2`

## Extension Type

Instance extension

## Registered Extension Number

120

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

## Contact

- James Jones [@cubanismo](#)

## Last Modified Date

2017-02-27

## IP Status

No known IP claims.

## Contributors

- Ian Elliott, Google
- James Jones, NVIDIA
- Alon Or-bach, Samsung

This extension provides new entry points to query device surface capabilities in a way that can be easily extended by other extensions, without introducing any further entry points. This extension can be considered the [VK\\_KHR\\_surface](#) equivalent of the [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#) extension.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_SURFACE\\_INFO\\_2\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_SURFACE\\_CAPABILITIES\\_2\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_SURFACE\\_FORMAT\\_2\\_KHR](#)

## New Enums

None.

## New Structures

- [VkPhysicalDeviceSurfaceInfo2KHR](#)
- [VkSurfaceCapabilities2KHR](#)
- [VkSurfaceFormat2KHR](#)

## New Functions

- [vkGetPhysicalDeviceSurfaceCapabilities2KHR](#)
- [vkGetPhysicalDeviceSurfaceFormats2KHR](#)

## Issues

1) What should this extension be named?

**RESOLVED:** [VK\\_KHR\\_get\\_surface\\_capabilities2](#). Other alternatives:

- [VK\\_KHR\\_surface2](#)
- One extension, combining a separate display-specific query extension.

2) Should additional WSI query functions be extended?

**RESOLVED:**

- [vkGetPhysicalDeviceSurfaceCapabilitiesKHR](#): Yes. The need for this motivated the extension.
- [vkGetPhysicalDeviceSurfaceSupportKHR](#): No. Currently only has boolean output. Extensions should instead extend [vkGetPhysicalDeviceSurfaceCapabilities2KHR](#).
- [vkGetPhysicalDeviceSurfaceFormatsKHR](#): Yes.
- [vkGetPhysicalDeviceSurfacePresentModesKHR](#): No. Recent discussion concluded this introduced too much variability for applications to deal with. Extensions should instead extend [vkGetPhysicalDeviceSurfaceCapabilities2KHR](#).
- [vkGetPhysicalDeviceXlibPresentationSupportKHR](#): Not in this extension.
- [vkGetPhysicalDeviceXcbPresentationSupportKHR](#): Not in this extension.
- [vkGetPhysicalDeviceWaylandPresentationSupportKHR](#): Not in this extension.
- [vkGetPhysicalDeviceWin32PresentationSupportKHR](#): Not in this extension.

## Version History

- Revision 1, 2017-02-27 (James Jones)
  - Initial draft.

## VK\_KHR\_image\_format\_list

### Name String

[VK\\_KHR\\_image\\_format\\_list](#)

## Extension Type

Device extension

## Registered Extension Number

148

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Jason Ekstrand [@jekstrand](#)

## Last Modified Date

2017-03-20

## IP Status

No known IP claims.

## Contributors

- Jason Ekstrand, Intel
- Jan-Harald Fredriksen, ARM
- Jeff Bolz, NVIDIA
- Jeff Leger, Qualcomm
- Neil Henning, Codeplay

On some implementations, setting the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` on image creation can cause access to that image to perform worse than an equivalent image created without `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` because the implementation does not know what view formats will be paired with the image.

This extension allows an application to provide the list of all formats that **can** be used with an image when it is created. The implementation may then be able to create a more efficient image that supports the subset of formats required by the application without having to support all formats in the format compatibility class of the image format.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO_KHR`

## New Enums

None.

## New Structs

- [VkImageFormatListCreateInfoKHR](#)

## New Functions

None.

## Issues

## Version History

- Revision 1, 2017-03-20 (Jason Ekstrand)
  - Initial revision

# VK\_KHR\_imageless\_framebuffer

## Name String

`VK_KHR_imageless_framebuffer`

## Extension Type

Device extension

## Registered Extension Number

109

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_maintenance2](#)
- Requires [VK\\_KHR\\_image\\_format\\_list](#)

## Contact

- Tobias Hector [@tobias](#)

## Last Modified Date

2018-12-14

## Contributors

- Tobias Hector
- Graham Wihlidal

This extension allows framebuffers to be created without the need for creating images first, allowing more flexibility in how they are used, and avoiding the need for many of the confusing compatibility rules.

Framebuffers are now created with a small amount of additional metadata about the image views that will be used in [VkFramebufferAttachmentsCreateInfoKHR](#), and the actual image views are provided at render pass begin time via [VkRenderPassAttachmentBeginInfoKHR](#).

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO_KHR`
- Extending [VkFramebufferCreateFlagBits](#):
  - `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR`

## New Structures

- [VkPhysicalDeviceImagelessFramebufferFeaturesKHR](#)
- [VkFramebufferAttachmentsCreateInfoKHR](#)
- [VkFramebufferAttachmentImageInfoKHR](#)
- [VkRenderPassAttachmentBeginInfoKHR](#)

## Version History

- Revision 1, 2018-12-14 (Tobias Hector)
  - Internal revisions

## **VK\_KHR\_incremental\_present**

### Name String

`VK_KHR_incremental_present`

### Extension Type

Device extension

### Registered Extension Number

85

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

- Requires [VK\\_KHR\\_swapchain](#)

## Contact

- Ian Elliott [Qianelliottus](#)

## Last Modified Date

2016-11-02

## IP Status

No known IP claims.

## Contributors

- Ian Elliott, Google
- Jesse Hall, Google
- Alon Or-bach, Samsung
- James Jones, NVIDIA
- Daniel Rakos, AMD
- Ray Smith, ARM
- Mika Isojarvi, Google
- Jeff Juliano, NVIDIA
- Jeff Bolz, NVIDIA

This device extension extends [vkQueuePresentKHR](#), from the [VK\\_KHR\\_swapchain](#) extension, allowing an application to specify a list of rectangular, modified regions of each image to present. This should be used in situations where an application is only changing a small portion of the presentable images within a swapchain, since it enables the presentation engine to avoid wasting time presenting parts of the surface that have not changed.

This extension is leveraged from the [EGL\\_KHR\\_swap\\_buffers\\_with\\_damage](#) extension.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PRESENT\\_REGIONS\\_KHR](#)

## New Enums

None.

## New Structures

- [VkRectLayerKHR](#)
- [VkPresentRegionKHR](#)
- [VkPresentRegionsKHR](#)

## New Functions

None.

## Examples

None.

## Issues

1) How should we handle stereoscopic-3D swapchains? We need to add a layer for each rectangle. One approach is to create another struct containing the [VkRect2D](#) plus layer, and have [VkPresentRegionsKHR](#) point to an array of that struct. Another approach is to have two parallel arrays, [pRectangles](#) and [pLayers](#), where [pRectangles\[i\]](#) and [pLayers\[i\]](#) must be used together. Which approach should we use, and if the array of a new structure, what should that be called?

**RESOLVED:** Create a new structure, which is a [VkRect2D](#) plus a layer, and will be called [VkRectLayerKHR](#).

2) Where is the origin of the [VkRectLayerKHR](#)?

**RESOLVED:** The upper left corner of the presentable image(s) of the swapchain, per the definition of framebuffer coordinates.

3) Does the rectangular region, [VkRectLayerKHR](#), specify pixels of the swapchain's image(s), or of the surface?

**RESOLVED:** Of the image(s). Some presentation engines may scale the pixels of a swapchain's image(s) to the size of the surface. The size of the swapchain's image(s) will be consistent, where the size of the surface may vary over time.

4) What if all of the rectangles for a given swapchain contain a width and/or height of zero?

**RESOLVED:** The application is indicating that no pixels changed since the last present. The presentation engine may use such a hint and not update any pixels for the swapchain. However, all other semantics of [vkQueuePresentKHR](#) must still be honored, including waiting for semaphores to signal.

## Version History

- Revision 1, 2016-11-02 (Ian Elliott)
  - Internal revisions

## **VK\_KHR\_performance\_query**

### **Name String**

`VK_KHR_performance_query`

### **Extension Type**

Device extension

### **Registered Extension Number**

117

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### **Contact**

- Alon Or-bach [@alonorbach](#)

### **Last Modified Date**

2019-10-08

### **IP Status**

No known IP claims.

### **Contributors**

- Jesse Barker, Unity Technologies
- Kenneth Benzie, Codeplay
- Jan-Harald Fredriksen, ARM
- Jeff Leger, Qualcomm
- Jesse Hall, Google
- Tobias Hector, AMD
- Neil Henning, Codeplay
- Baldur Karlsson
- Lionel Landwerlin, Intel
- Peter Lohrmann, AMD
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Niklas Smedberg, Unity Technologies
- Igor Ostrowski, Intel

The `VK_KHR_performance_query` extension adds a mechanism to allow querying of performance counters for use in applications and by profiling tools.

Each queue family can expose counters that can be enabled on a queue of that family. We extend `VkQueryType` to add a new query type for performance queries, and chain a structure on `VkQueryPoolCreateInfo` to specify the performance queries to enable.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR`
  - `VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR`
  - `VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR`
  - `VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR`
- Extending `VkQueryType`:
  - `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`

## New Enums

- `VkPerformanceCounterScopeKHR`
- `VkPerformanceCounterStorageKHR`
- `VkPerformanceCounterUnitKHR`
- `VkPerformanceCounterDescriptionFlagBitsKHR`
- `VkAcquireProfilingLockFlagBitsKHR`

## New Structures

- `VkPhysicalDevicePerformanceQueryFeaturesKHR`
- `VkPhysicalDevicePerformanceQueryPropertiesKHR`
- `VkPerformanceCounterKHR`
- `VkPerformanceCounterDescriptionKHR`
- `VkPerformanceCounterDescriptionFlagsKHR`
- `VkQueryPoolPerformanceCreateInfoKHR`
- `VkPerformanceCounterResultKHR`
- `VkAcquireProfilingLockInfoKHR`

- [VkAcquireProfilingLockFlagsKHR](#)
- [VkPerformanceQuerySubmitInfoKHR](#)

## New Functions

- [vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR](#)
- [vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR](#)
- [vkAcquireProfilingLockKHR](#)
- [vkReleaseProfilingLockKHR](#)

## Issues

1) Should this extension include a mechanism to begin a query in command buffer *A* and end the query in command buffer *B*?

**RESOLVED** No - queries are tied to command buffer creation and thus have to be encapsulated within a single command buffer.

2) Should this extension include a mechanism to begin and end queries globally on the queue, not using the existing command buffer commands?

**RESOLVED** No - for the same reasoning as the resolution of 1).

3) Should this extension expose counters that require multiple passes?

**RESOLVED** Yes - users should re-submit a command buffer with the same commands in it multiple times, specifying the pass to count as the query parameter in [VkPerformanceQuerySubmitInfoKHR](#).

4) How to handle counters across parallel workloads?

**RESOLVED** In the spirit of Vulkan, a counter description flag [VK\\_PERFORMANCE\\_COUNTER\\_DESCRIPTION\\_CONCURRENTLY\\_IMPACTED\\_KHR](#) denotes that the accuracy of a counter result is affected by parallel workloads.

5) How to handle secondary command buffers?

**RESOLVED** Secondary command buffers inherit any counter pass index specified in the parent primary command buffer. Note: this is no longer an issue after change from issue 10 resolution

6) What commands does the profiling lock have to be held for?

**RESOLVED** For any command buffer that is being queried with a performance query pool, the profiling lock **must** be held while that command buffer is in the *recording*, *executable*, or *pending state*.

7) Should we support [vkCmdCopyQueryPoolResults](#)?

**RESOLVED** Yes.

8) Should we allow performance queries to interact with multiview?

**RESOLVED** Yes, but the performance queries must be performed once for each pass per view.

9) Should a queryCount > 1 be usable for performance queries?

**RESOLVED** Yes. Some vendors will have costly performance counter query pool creation, and would rather if a certain set of counters were to be used multiple times that a queryCount > 1 can be used to amortize the instantiation cost.

10) Should we introduce an indirect mechanism to set the counter pass index?

**RESOLVED** Specify the counter pass index at submit time instead to avoid requiring re-recording of command buffers when multiple counter passes needed.

## Examples

The following example shows how to find what performance counters a queue family supports, setup a query pool to record these performance counters, how to add the query pool to the command buffer to record information, and how to get the results from the query pool.

```
// A previously created physical device
VkPhysicalDevice physicalDevice;

// One of the queue families our device supports
uint32_t queueFamilyIndex;

uint32_t counterCount;

// Get the count of counters supported
vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(
    physicalDevice,
    queueFamilyIndex,
    &counterCount,
    NULL,
    NULL);

VkPerformanceCounterKHR* counters =
    malloc(sizeof(VkPerformanceCounterKHR) * counterCount);
VkPerformanceCounterDescriptionKHR* counterDescriptions =
    malloc(sizeof(VkPerformanceCounterDescriptionKHR) * counterCount);

// Get the counters supported
vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(
    physicalDevice,
    queueFamilyIndex,
    &counterCount,
    counters,
    counterDescriptions);

// Try to enable the first 8 counters
uint32_t enabledCounters[8];
```

```

const uint32_t enabledCounterCount = min(counterCount, 8));

for (uint32_t i = 0; i < enabledCounterCount; i++) {
    enabledCounters[i] = i;
}

// A previously created device that had the performanceCounterQueryPools feature
// set to VK_TRUE
VkDevice device;

VkQueryPoolPerformanceCreateInfoKHR performanceQueryCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR,
    NULL,

    // Specify the queue family that this performance query is performed on
    queueFamilyIndex,

    // The number of counters to enable
    enabledCounterCount,

    // The array of indices of counters to enable
    enabledCounters
};

// Get the number of passes our counters will require.
uint32_t numPasses;

vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR(
    physicalDevice,
    &performanceQueryCreateInfo,
    &numPasses);

VkQueryPoolCreateInfo queryPoolCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO,
    &performanceQueryCreateInfo,
    0,

    // Using our new query type here
    VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR,

    1,
    0
};

VkQueryPool queryPool;

VkResult result = vkCreateQueryPool(
    device,

```

```

    &queryPoolCreateInfo,
    NULL,
    &queryPool);

assert(VK_SUCCESS == result);

// A queue from queueFamilyIndex
VkQueue queue;

// A command buffer we want to record counters on
VkCommandBuffer commandBuffer;

VkCommandBufferBeginInfo commandBufferBeginInfo = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    NULL,
    0,
    NULL
};

VkAcquireProfilingLockInfoKHR lockInfo = {
    VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR,
    NULL,
    0,
    UINT64_MAX // Wait forever for the lock
};

// Acquire the profiling lock before we record command buffers
// that will use performance queries

result = vkAcquireProfilingLockKHR(device, &lockInfo);

assert(VK_SUCCESS == result);

result = vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);

assert(VK_SUCCESS == result);

vkCmdResetQueryPool(
    commandBuffer,
    queryPool,
    0,
    1);

vkCmdBeginQuery(
    commandBuffer,
    queryPool,
    0,
    0);

// Perform the commands you want to get performance information on
// ...

```

```

// Perform a barrier to ensure all previous commands were complete before
// ending the query
vkCmdPipelineBarrier(commandBuffer,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    0,
    0,
    NULL,
    0,
    NULL,
    0,
    NULL);

```

```

vkCmdEndQuery(
    commandBuffer,
    queryPool,
    0);

```

```

result = vkEndCommandBuffer(commandBuffer);

assert(VK_SUCCESS == result);

for (uint32_t counterPass = 0; counterPass < numPasses; counterPass++) {

    VkPerformanceQuerySubmitInfoKHR performanceQuerySubmitInfo = {
        VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR,
        NULL,
        counterPass
    };

    // Submit the command buffer and wait for its completion
    // ...
}

// Release the profiling lock after the command buffer is no longer in the
// pending state.
vkReleaseProfilingLockKHR(device);

result = vkResetCommandBuffer(commandBuffer, 0);

assert(VK_SUCCESS == result);

// Create an array to hold the results of all counters
VkPerformanceCounterResultKHR* recordedCounters = malloc(
    sizeof(VkPerformanceCounterResultKHR) * enabledCounterCount);

result = vkGetQueryPoolResults(
    device,
    queryPool,

```

```

0,
1,
sizeof(VkPerformanceCounterResultKHR) * enabledCounterCount,
recordedCounters,
sizeof(VkPerformanceCounterResultKHR),
NULL);

// recordedCounters is filled with our counters, we'll look at one for posterity
switch (counters[0].storage) {
    case VK_PERFORMANCE_COUNTER_STORAGE_INT32:
        // use recordCounters[0].int32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_INT64:
        // use recordCounters[0].int64 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_UINT32:
        // use recordCounters[0].uint32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_UINT64:
        // use recordCounters[0].uint64 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32:
        // use recordCounters[0].float32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64:
        // use recordCounters[0].float64 to get at the counter result!
        break;
}

```

## Version History

- Revision 1, 2019-10-08

## VK\_KHR\_pipeline\_executable\_properties

### Name String

`VK_KHR_pipeline_executable_properties`

### Extension Type

Device extension

### Registered Extension Number

270

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Jason Ekstrand [@jekstrand](#)

## Last Modified Date

2019-05-28

## IP Status

No known IP claims.

## Interactions and External Dependencies

### Contributors

- Jason Ekstrand, Intel
- Ian Romanick, Intel
- Kenneth Graunke, Intel
- Baldur Karlsson, Valve
- Jesse Hall, Google
- Jeff Bolz, Nvidia
- Piers Daniel, Nvidia
- Tobias Hector, AMD
- Jan-Harald Fredriksen, ARM
- Tom Olson, ARM
- Daniel Koch, Nvidia
- Spencer Fricke, Samsung

When a pipeline is created, its state and shaders are compiled into zero or more device-specific executables, which are used when executing commands against that pipeline. This extension adds a mechanism to query properties and statistics about the different executables produced by the pipeline compilation process. This is intended to be used by debugging and performance tools to allow them to provide more detailed information to the user. Certain compile-time shader statistics provided through this extension may be useful to developers for debugging or performance analysis.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_PIPELINE\\_EXECUTABLE\\_PROPERTIES\\_FEATURES\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_INFO\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_EXECUTABLE\\_PROPERTIES\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_EXECUTABLE\\_INFO\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_EXECUTABLE\\_STATISTIC\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_EXECUTABLE\\_INTERNAL\\_REPRESENTATION\\_KHR](#)
- Extending [VkPipelineCreateFlagBits](#):

- VK\_PIPELINE\_CREATE\_CAPTURE\_STATISTICS\_BIT\_KHR
- VK\_PIPELINE\_CREATE\_CAPTURE\_INTERNAL REPRESENTATIONS\_BIT\_KHR

## New Enums

- [VkPipelineExecutableStatisticFormatKHR](#)

## New Structures

- [VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR](#)
- [VkPipelineInfoKHR](#)
- [VkPipelineExecutablePropertiesKHR](#)
- [VkPipelineExecutableInfoKHR](#)
- [VkPipelineExecutableStatisticValueKHR](#)
- [VkPipelineExecutableStatisticKHR](#)
- [VkPipelineExecutableInternalRepresentationKHR](#)

## New Functions

- [vkGetPipelineExecutablePropertiesKHR](#)
- [vkGetPipelineExecutableStatisticsKHR](#)
- [vkGetPipelineExecutableInternalRepresentationsKHR](#)

## Issues

1) What should we call the pieces of the pipeline which are produced by the compilation process and about which you can query properties and statistics?

**RESOLVED:** Call them "executables". The name "binary" was used in early drafts of the extension but it was determined that "pipeline binary" could have a fairly broad meaning (such as a binary serialized form of an entire pipeline) and was too big of a namespace for the very specific needs of this extension.

## Version History

- Revision 1, 2019-05-28 (Jason Ekstrand)
  - Initial draft

## VK\_KHR\_push\_descriptor

### Name String

VK\_KHR\_push\_descriptor

### Extension Type

Device extension

## Registered Extension Number

81

## Revision

2

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Jeff Bolz [@jeffbolznv](#)

## Last Modified Date

2017-09-12

## IP Status

No known IP claims.

## Contributors

- Jeff Bolz, NVIDIA
- Michael Worcester, Imagination Technologies

This extension allows descriptors to be written into the command buffer, while the implementation is responsible for managing their memory. Push descriptors may enable easier porting from older APIs and in some cases can be more efficient than writing descriptors into descriptor sets.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_PUSH\\_DESCRIPTOR\\_PROPERTIES\\_KHR](#)
- Extending [VkDescriptorSetLayoutCreateFlagBits](#)
  - [VK\\_DESCRIPTOR\\_SET\\_LAYOUT\\_CREATE\\_PUSH\\_DESCRIPTOR\\_BIT\\_KHR](#)
- Extending [VkDescriptorUpdateTemplateType](#)
  - [VK\\_DESCRIPTOR\\_UPDATE\\_TEMPLATE\\_TYPE\\_PUSH\\_DESCRIPTORS\\_KHR](#)

## New Enums

None.

## New Structures

- [VkPhysicalDevicePushDescriptorPropertiesKHR](#)

## New Functions

- [vkCmdPushDescriptorSetKHR](#)
- [vkCmdPushDescriptorSetWithTemplateKHR](#)

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2016-10-15 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2017-09-12 (Tobias Hector)
  - Added interactions with Vulkan 1.1

## VK\_KHR\_sampler\_mirror\_clamp\_to\_edge

### Name String

`VK_KHR_sampler_mirror_clamp_to_edge`

### Extension Type

Device extension

### Registered Extension Number

15

### Revision

3

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Tobias Hector [@tobski](#)

### Last Modified Date

2019-08-17

## Contributors

- Tobias Hector, Imagination Technologies
- Jon Leech, Khronos

`VK_KHR_sampler_mirror_clamp_to_edge` extends the set of sampler address modes to include an additional mode (`VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE`) that effectively uses a texture map twice as large as the original image in which the additional half of the new image is a mirror image of the original image.

This new mode relaxes the need to generate images whose opposite edges match by using the original image to generate a matching “mirror image”. This mode allows the texture to be mirrored only once in the negative s, t, and r directions.

## New Enum Constants

- Extending `VkSamplerAddressMode`:
  - `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE`

## Example

Creating a sampler with the new address mode in each dimension

```
VkSamplerCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO // sType
    // Other members set to application-desired values
};

createInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE;
createInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE;
createInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE;

VkSampler sampler;
VkResult result = vkCreateSampler(
    device,
    &createInfo,
    &sampler);
```

## Issues

1) Why are both KHR and core versions of the `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` token present?

**RESOLVED:** This functionality was intended to be required in Vulkan 1.0. We realized shortly before public release that not all implementations could support it, and moved the functionality into an optional extension, but did not apply the KHR extension suffix. Adding a KHR-suffixed alias of the non-suffixed enum has been done to comply with our own naming rules.

In a related change, before spec revision 1.1.121 this extension was hardwiring into the spec Makefile so it was always included with the Specification, even in the core-only versions. This has now been reverted, and it is treated as any other extension.

## Version History

- Revision 1, 2016-02-16 (Tobias Hector)
  - Initial draft
- Revision 2, 2019-08-14 (Jon Leech)
  - Add KHR-suffixed alias of non-suffixed enum.
- Revision 3, 2019-08-17 (Jon Leech)
  - Add an issue explaining the reason for the extension API not being suffixed with KHR.

## **VK\_KHR\_separate\_depth\_stencil\_layouts**

### Name String

`VK_KHR_separate_depth_stencil_layouts`

### Extension Type

Device extension

### Registered Extension Number

242

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_create_renderpass2`

### Contact

- Piers Daniell [Opdaniell-nv](#)

### Last Modified Data

2019-06-25

### Contributors

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA
- Jesse Barker, Unity
- Tobias Hector, AMD

This extension allows image memory barriers for depth/stencil images to have just one of the

`VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` aspect bits set, rather than require both. This allows their layouts to be set independently. To support depth/stencil images with different layouts for the depth and stencil aspects, the depth/stencil attachment interface has been updated to support a separate layout for stencil.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT_KHR`
  - `VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT_KHR`
- Extending [VkImageLayout](#):
  - `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR`
  - `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR`
  - `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR`
  - `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR`

## New Enums

None.

## New Structures

- Extending [VkPhysicalDeviceFeatures2](#):
  - [VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR](#)
- Extending [VkAttachmentReference2KHR](#):
  - [VkAttachmentReferenceStencilLayoutKHR](#)
- Extending [VkAttachmentDescription2KHR](#):
  - [VkAttachmentDescriptionStencilLayoutKHR](#)

## New Functions

None.

## New SPIR-V Capabilities

None.

## Issues

None.

## Version History

- Revision 1, 2019-06-25 (Piers Daniell)
  - Internal revisions

## VK\_KHR\_shader\_atomic\_int64

### Name String

`VK_KHR_shader_atomic_int64`

### Extension Type

Device extension

### Registered Extension Number

181

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Aaron Hagan [@ahagan](#)

### Last Modified Date

2018-07-05

### Interactions and External Dependencies

- This extension requires the `GL_ARB_gpu_shader_int64` and `GL_EXT_shader_atomic_int64` extensions for GLSL source languages.

### Contributors

- Aaron Hagan, AMD
- Daniel Rakos, AMD
- Jeff Bolz, NVIDIA
- Neil Henning, Codeplay

This extension advertises the SPIR-V **Int64Atomics** capability for Vulkan, which allows a shader to contain 64-bit atomic operations on signed and unsigned integers. The supported operations include OpAtomicMin, OpAtomicMax, OpAtomicAnd, OpAtomicOr, OpAtomicXor, OpAtomicAdd, OpAtomicExchange, and OpAtomicCompareExchange.

## New Enum Constants

- Extending `VkStructureType`:

- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES_KHR`

## New SPIR-V Capabilities

- `Int64Atomics`

## New Structures

- [VkPhysicalDeviceShaderAtomicInt64FeaturesKHR](#)

## Version History

- Revision 1, 2018-07-05 (Aaron Hagan)
  - Internal revisions

## `VK_KHR_shader_clock`

### Name String

`VK_KHR_shader_clock`

### Extension Type

Device extension

### Registered Extension Number

182

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Aaron Hagan [@ahagan](#)

### Last Modified Date

2019-4-25

### IP Status

No known IP claims.

### Interactions and External Dependencies

- This extension requires `SPV_KHR_shader_clock`.
- This extension enables `ARB_shader_clock` for GLSL source languages.
- This extension enables `EXT_shader_realtime_clock` for GLSL source languages.

## Contributors

- Aaron Hagan, AMD
- Daniel Koch, NVIDIA

This extension advertises the SPIR-V `ShaderClockKHR` capability for Vulkan, which allows a shader to query a real-time or monotonically incrementing counter at the subgroup level or across the device level. The two valid SPIR-V scopes for `OpReadClockKHR` are `Subgroup` and `Device`.

When using GLSL source-based shading languages, the `clockRealtime*EXT()` timing functions map to the `OpReadClockKHR` instruction with a scope of `Device`, and the `clock*ARB()` timing functions map to the `OpReadClockKHR` instruction with a scope of `Subgroup`.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CLOCK_FEATURES_KHR`

## New SPIR-V Capabilities

- `ShaderClockKHR`

## New Structures

- `VkPhysicalDeviceShaderClockFeaturesKHR`

## Version History

- Revision 1, 2019-4-25 (Aaron Hagan)
  - Initial revision

## `VK_KHR_shader_float16_int8`

### Name String

`VK_KHR_shader_float16_int8`

### Extension Type

Device extension

### Registered Extension Number

83

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Alexander Galazin [@legal-arm](#)

## Last Modified Date

2018-03-07

## IP Status

No known IP claims.

## Interactions and External Dependencies

- This extension interacts with [VK\\_KHR\\_8bit\\_storage](#)
- This extension interacts with [VK\\_KHR\\_16bit\\_storage](#)
- This extension interacts with [VK\\_KHR\\_shader\\_float\\_controls](#)

## Contributors

- Alexander Galazin, Arm
- Jan-Harald Fredriksen, Arm
- Jeff Bolz, NVIDIA
- Graeme Leese, Broadcom
- Daniel Rakos, AMD

## Description

The [VK\\_KHR\\_shader\\_float16\\_int8](#) extension allows use of 16-bit floating-point types and 8-bit integer types in shaders for arithmetic operations.

It introduces two new optional features `shaderFloat16` and `shaderInt8` which directly map to the `Float16` and the `Int8` SPIR-V capabilities. The [VK\\_KHR\\_shader\\_float16\\_int8](#) extension also specifies precision requirements for half-precision floating-point SPIR-V operations. This extension does not enable use of 8-bit integer types or 16-bit floating-point types in any `shader input` and `output interfaces` and therefore does not supersede the [VK\\_KHR\\_8bit\\_storage](#) or [VK\\_KHR\\_16bit\\_storage](#) extensions.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_SHADER\\_FLOAT16\\_INT8\\_FEATURES\\_KHR](#)

## New Structures

- [VkPhysicalDeviceShaderFloat16Int8FeaturesKHR](#)

## New Functions

- None

## Version History

- Revision 1, 2018-03-07 (Alexander Galazin)
  - Initial draft

## VK\_KHR\_shader\_float\_controls

### Name String

`VK_KHR_shader_float_controls`

### Extension Type

Device extension

### Registered Extension Number

198

### Revision

4

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Alexander Galazin [@legal-arm](#)

### Last Modified Date

2018-09-11

### IP Status

No known IP claims.

### Interactions and External Dependencies

- This extension requires `SPV_KHR_float_controls`

### Contributors

- Alexander Galazin, Arm
- Jan-Harald Fredriksen, Arm
- Jeff Bolz, NVIDIA
- Graeme Leese, Broadcom
- Daniel Rakos, AMD

## Description

The `VK_KHR_shader_float_controls` extension enables efficient use of floating-point computations through the ability to query and override the implementation's default behavior for rounding modes, denormals, signed zero, and infinity.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES_KHR`

## New Enums

- None

## New Structures

- `VkPhysicalDeviceFloatControlsPropertiesKHR`

## New Functions

- None

## New SPIR-V Capabilities

- `DenormPreserve`
- `DenormFlushToZero`
- `SignedZeroInfNanPreserve`
- `RoundingModeRTE`
- `RoundingModeRTZ`

## Issues

1) Which instructions must flush denorms?

**RESOLVED:** Only floating-point conversion, floating-point arithmetic, floating-point relational (except `OpIsNaN`, `OpIsInf`), and floating-point GLSL.std.450 extended instructions must flush denormals.

2) What is the denorm behavior for intermediate results?

**RESOLVED:** When a SPIR-V instruction is implemented as a sequence of other instructions: - in the `DenormFlushToZero` execution mode the intermediate instructions may flush denormals, the final result of the sequence **must** not be denormal. - in the `DenormPreserve` execution mode denormals must be preserved throughout the whole sequence.

3) Do denorm and rounding mode controls apply to `OpSpecConstantOp`?

**RESOLVED:** Yes, except when the opcode is `OpQuantizeToF16`.

4) The SPIR-V specification says that `OpConvertFToU` and `OpConvertFToS` unconditionally round towards zero. Do the rounding mode controls specified through the execution modes apply to them?

**RESOLVED:** No, these instructions unconditionally round towards zero.

5) Do any of the "Pack" GLSL.std.450 instructions count as conversion instructions and have the rounding mode apply?

**RESOLVED:** No, only instructions listed in the section "3.32.11. Conversion Instructions" of the SPIR-V specification count as conversion instructions.

6) When using inf/nan-ignore mode, what is expected of `OpIsNan` and `OpIsInf`?

**RESOLVED:** These instructions must always accurately detect inf/nan if it is passed to them.

## Version 4 API incompatibility

The original versions of `VK_KHR_shader_controls` shipped with booleans named "separateDenormSettings" and "separateRoundingModeSettings", which at first glance could've indicated "they can all independently set, or not". However the spec language as written indicated that the 32-bit value could always be set independently, and only the 16- and 64-bit controls needed to be the same if these values were `VK_FALSE`.

As a result of this slight disparity, and lack of test coverage for this facet of the extension, we ended up with two different behaviors in the wild, where some implementations worked as written, and others worked based on the naming. As these are hard limits in hardware with reasons for exposure as written, it was not possible to standardise on a single way to make this work within the existing API.

No known users of this part of the extension exist in the wild, and as such the Vulkan WG took the unusual step of retroactively changing the once boolean value into a tri-state enum, breaking source compatibility. This was however done in such a way as to retain ABI compatibility, in case any code using this did exist; with the numerical values 0 and 1 retaining their original specified meaning, and a new value signifying the additional "all need to be set together" state. If any applications exist today, compiled binaries will continue to work as written in most cases, but will need changes before the code can be recompiled.

## Version History

- Revision 4, 2019-06-18 (Tobias Hector)
  - Modified settings restrictions, see [Version 4 API incompatibility](#)
- Revision 3, 2018-09-11 (Alexander Galazin)
  - Minor restructuring
- Revision 2, 2018-04-17 (Alexander Galazin)
  - Added issues and resolutions
- Revision 1, 2018-04-11 (Alexander Galazin)

- Initial draft

## **VK\_KHR\_shader\_subgroup\_extended\_types**

### **Name String**

`VK_KHR_shader_subgroup_extended_types`

### **Extension Type**

Device extension

### **Registered Extension Number**

176

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.1

### **Contact**

- Neil Henning [Osheredom](#)

### **Last Modified Date**

2019-01-08

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

None.

### **Contributors**

- Jeff Bolz, NVIDIA
- Jan-Harald Fredriksen, Arm
- Neil Henning, AMD
- Daniel Koch, NVIDIA
- Jeff Leger, Qualcomm
- Graeme Leese, Broadcom
- David Neto, Google
- Daniel Rakos, AMD

This extension enables the Non Uniform Group Operations in SPIR-V to support 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES_KHR`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR](#)

## New Functions

None.

## New Built-In Variables

None.

## New SPIR-V Capabilities

None.

## Issues

None.

## Version History

- Revision 1, 2019-01-08 (Neil Henning)
  - Initial draft

## VK\_KHR\_shared\_presentable\_image

### Name String

`VK_KHR_shared_presentable_image`

### Extension Type

Device extension

### Registered Extension Number

112

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_swapchain`
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_get_surface_capabilities2`

## Contact

- Alon Or-bach [@alonorbach](#)

## Last Modified Date

2017-03-20

## IP Status

No known IP claims.

## Contributors

- Alon Or-bach, Samsung Electronics
- Ian Elliott, Google
- Jesse Hall, Google
- Pablo Ceballos, Google
- Chris Forbes, Google
- Jeff Juliano, NVIDIA
- James Jones, NVIDIA
- Daniel Rakos, AMD
- Tobias Hector, Imagination Technologies
- Graham Connor, Imagination Technologies
- Michael Worcester, Imagination Technologies
- Cass Everitt, Oculus
- Johannes Van Waveren, Oculus

This extension extends `VK_KHR_swapchain` to enable creation of a shared presentable image. This allows the application to use the image while the presentation engine is accessing it, in order to reduce the latency between rendering and presentation.

## New Object Types

None.

## New Enum Constants

- Extending [VkPresentModeKHR](#):
  - `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR`
  - `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`
- Extending [VkImageLayout](#):
  - `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR`

## New Enums

None.

## New Structures

- [VkSharedPresentSurfaceCapabilitiesKHR](#)

## New Functions

- [vkGetSwapchainStatusKHR](#)

## Issues

1) Should we allow a Vulkan WSI swapchain to toggle between normal usage and shared presentation usage?

**RESOLVED:** No. WSI swapchains are typically recreated with new properties instead of having their properties changed. This can also save resources, assuming that fewer images are needed for shared presentation, and assuming that most VR applications do not need to switch between normal and shared usage.

2) Should we have a query for determining how the presentation engine refresh is triggered?

**RESOLVED:** Yes. This is done via which presentation modes a surface supports.

3) Should the object representing a shared presentable image be an extension of a [VkSwapchainKHR](#) or a separate object?

**RESOLVED:** Extension of a swapchain due to overlap in creation properties and to allow common functionality between shared and normal presentable images and swapchains.

4) What should we call the extension and the new structures it creates?

**RESOLVED:** Shared presentable image / shared present.

5) Should the `minImageCount` and `presentMode` values of the [VkSwapchainCreateInfoKHR](#) be ignored, or required to be compatible values?

**RESOLVED:** `minImageCount` must be set to 1, and `presentMode` should be set to either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`.

6) What should the layout of the shared presentable image be?

**RESOLVED:** After acquiring the shared presentable image, the application must transition it to the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` layout prior to it being used. After this initial transition, any image usage that was requested during swapchain creation **can** be performed on the image without layout transitions being performed.

7) Do we need a new API for the trigger to refresh new content?

**RESOLVED:** `vkQueuePresentKHR` to act as API to trigger a refresh, as will allow combination with other compatible extensions to `vkQueuePresentKHR`.

8) How should an application detect a `VK_ERROR_OUT_OF_DATE_KHR` error on a swapchain using the `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` present mode?

**RESOLVED:** Introduce `vkGetSwapchainStatusKHR` to allow applications to query the status of a swapchain using a shared presentation mode.

9) What should subsequent calls to `vkQueuePresentKHR` for `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` swapchains be defined to do?

**RESOLVED:** State that implementations may use it as a hint for updated content.

10) Can the ownership of a shared presentable image be transferred to a different queue?

**RESOLVED:** No. It is not possible to transfer ownership of a shared presentable image obtained from a swapchain created using `VK_SHARING_MODE_EXCLUSIVE` after it has been presented.

11) How should `vkQueueSubmit` behave if a command buffer uses an image from an `VK_ERROR_OUT_OF_DATE_KHR` swapchain?

**RESOLVED:** `vkQueueSubmit` is expected to return the `VK_ERROR_DEVICE_LOST` error.

12) Can Vulkan provide any guarantee on the order of rendering, to enable beam chasing?

**RESOLVED:** This could be achieved via use of render passes to ensure strip rendering.

## Version History

- Revision 1, 2017-03-20 (Alon Or-bach)
  - Internal revisions

## VK\_KHR\_spirv\_1\_4

### Name String

`VK_KHR_spirv_1_4`

### Extension Type

Device extension

## Registered Extension Number

237

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.1
- Requires [VK\\_KHR\\_shader\\_float\\_controls](#)

## Contact

- Jesse Hall [critsec](#)

## Last Modified Date

2019-04-01

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Requires SPIR-V 1.4.

## Contributors

- Alexander Galazin, Arm
- David Neto, Google
- Jesse Hall, Google
- John Kessenich, Google
- Neil Henning, AMD
- Tom Olson, Arm

This extension allows the use of SPIR-V 1.4 shader modules. SPIR-V 1.4's new features primarily make it an easier target for compilers from high-level languages, rather than exposing new hardware functionality.

SPIR-V 1.4 incorporates features that are also available separately as extensions. SPIR-V 1.4 shader modules do not need to enable those extensions with the [OpExtension](#) opcode, since they are integral parts of SPIR-V 1.4.

SPIR-V 1.4 introduces new floating point execution mode capabilities, also available via [SPV\\_KHR\\_float\\_controls](#). Implementations are not required to support all of these new capabilities; support can be queried using [VkPhysicalDeviceFloatControlsPropertiesKHR](#) from the [VK\\_KHR\\_shader\\_float\\_controls](#) extension.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## Issues

1. Should we have an extension specific to this SPIR-V version, or add a version-generic query for SPIR-V version? SPIR-V 1.4 doesn't need any other API changes.

RESOLVED: Just expose SPIR-V 1.4.

Most new SPIR-V versions introduce optionally-required capabilities or have implementation-defined limits, and would need more API and specification changes specific to that version to make them available in Vulkan. For example, to support the subgroup features added in SPIR-V 1.3 required introducing [VkPhysicalDeviceSubgroupProperties](#) to allow querying the supported subgroup operation categories, maximum supported subgroup size, etc. While we could expose the parts of a new SPIR-V version that don't need accompanying changes generically, we'll still end up writing extensions specific to each version for the remaining parts. Thus the generic mechanism won't reduce future spec-writing effort. In addition, making it clear which parts of a future version are supported by the generic mechanism and which can't be used without specific support would be difficult to get right ahead of time.

2. Can different stages of the same pipeline use shaders with different SPIR-V versions?

RESOLVED: Yes.

Mixing SPIR-V versions 1.0-1.3 in the same pipeline has not been disallowed, so it would be inconsistent to disallow mixing 1.4 with previous versions.. SPIR-V 1.4 does not introduce anything that should cause new difficulties here.

3. Must Vulkan extensions corresponding to SPIR-V extensions that were promoted to core in 1.4 be enabled in order to use that functionality in a SPIR-V 1.4 module?

RESOLVED: No, with caveats.

The SPIR-V 1.4 module does not need to declare the SPIR-V extensions, since the functionality is now part of core, so there is no need to enable the Vulkan extension that allows SPIR-V modules to declare the SPIR-V extension. However, when the functionality that is now core in SPIR-V 1.4 is optionally supported, the query for support is provided by a Vulkan extension, and that query can only be used if the extension is enabled.

This applies to any SPIR-V version; specifically for SPIR-V 1.4 this only applies to the functionality from `SPV_KHR_float_controls`, which was made available in Vulkan by `VK_KHR_shader_float_controls`. Even though the extension was promoted in SPIR-V 1.4, the capabilities are still optional in implementations that support `VK_KHR_spirv_1_4`.

A SPIR-V 1.4 module doesn't need to enable `SPV_KHR_float_controls` in order to use the capabilities, so if the application has *a priori* knowledge that the implementation supports the capabilities, it doesn't need to enable `VK_KHR_shader_float_controls`. However, if it doesn't have this knowledge and has to query for support at runtime, it must enable `VK_KHR_shader_float_controls` in order to use `VkPhysicalDeviceFloatControlsPropertiesKHR`.

## Version History

- Revision 1, 2019-04-01 (Jesse Hall)
  - Internal draft versions

## `VK_KHR_surface`

### Name String

`VK_KHR_surface`

### Extension Type

Instance extension

### Registered Extension Number

1

### Revision

25

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- James Jones [Qcubanismo](#)
- Ian Elliott [Qianelliottus](#)

### Last Modified Date

2016-08-25

### IP Status

No known IP claims.

## Contributors

- Patrick Doane, Blizzard
- Ian Elliott, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG
- Jason Ekstrand, Intel

The `VK_KHR_surface` extension is an instance extension. It introduces `VkSurfaceKHR` objects, which abstract native platform surface or window objects for use with Vulkan. It also provides a way to determine whether a queue family in a physical device supports presenting to particular surface.

Separate extensions for each platform provide the mechanisms for creating `VkSurfaceKHR` objects, but once created they may be used in this and other platform-independent extensions, in particular the `VK_KHR_swapchain` extension.

## New Object Types

- `VkSurfaceKHR`

## New Enum Constants

- Extending `VkResult`:
  - `VK_ERROR_SURFACE_LOST_KHR`
  - `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

## New Enums

- `VkSurfaceTransformFlagBitsKHR`
- `VkPresentModeKHR`
- `VkColorSpaceKHR`
- `VkCompositeAlphaFlagBitsKHR`

## New Structures

- `VkSurfaceCapabilitiesKHR`

- [VkSurfaceFormatKHR](#)

## New Functions

- [vkDestroySurfaceKHR](#)
- [vkGetPhysicalDeviceSurfaceSupportKHR](#)
- [vkGetPhysicalDeviceSurfaceCapabilitiesKHR](#)
- [vkGetPhysicalDeviceSurfaceFormatsKHR](#)
- [vkGetPhysicalDeviceSurfacePresentModesKHR](#)

## Examples

*Note*



The example code for the `VK_KHR_surface` and `VK_KHR_swapchain` extensions was removed from the appendix after revision 1.0.29. This WSI example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: <https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c>).

## Issues

1) Should this extension include a method to query whether a physical device supports presenting to a specific window or native surface on a given platform?

**RESOLVED:** Yes. Without this, applications would need to create a device instance to determine whether a particular window can be presented to. Knowing that a device supports presentation to a platform in general is not sufficient, as a single machine might support multiple seats, or instances of the platform that each use different underlying physical devices. Additionally, on some platforms, such as the X Window System, different drivers and devices might be used for different windows depending on which section of the desktop they exist on.

2) Should the `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, `vkGetPhysicalDeviceSurfaceFormatsKHR`, and `vkGetPhysicalDeviceSurfacePresentModesKHR` functions from `VK_KHR_swapchain` be modified to operate on physical devices and moved to this extension to implement the resolution of issue 1?

**RESOLVED:** No, separate query functions are needed, as the purposes served are similar but incompatible. The `vkGetPhysicalDeviceSurface*KHR` functions return information that could potentially depend on an initialized device. For example, the formats supported for presentation to the surface might vary depending on which device extensions are enabled. The query introduced to resolve issue 1 should be used only to query generic driver or platform properties. The physical device parameter is intended to serve only as an identifier rather than a stateful object.

3) Should Vulkan include support Xlib or XCB as the API for accessing the X Window System platform?

**RESOLVED:** Both. XCB is a more modern and efficient API, but Xlib usage is deeply ingrained in

many applications and likely will remain in use for the foreseeable future. Not all drivers necessarily need to support both, but including both as options in the core specification will probably encourage support, which should in turn ease adoption of the Vulkan API in older codebases. Additionally, the performance improvements possible with XCB likely will not have a measurable impact on the performance of Vulkan presentation and other minimal window system interactions defined here.

4) Should the GBM platform be included in the list of platform enums?

**RESOLVED:** Deferred, and will be addressed with a platform-specific extension to be written in the future.

## Version History

- Revision 1, 2015-05-20 (James Jones)
  - Initial draft, based on LunarG KHR spec, other KHR specs, patches attached to bugs.
- Revision 2, 2015-05-22 (Ian Elliott)
  - Created initial Description section.
  - Removed query for whether a platform requires the use of a queue for presentation, since it was decided that presentation will always be modeled as being part of the queue.
  - Fixed typos and other minor mistakes.
- Revision 3, 2015-05-26 (Ian Elliott)
  - Improved the Description section.
- Revision 4, 2015-05-27 (James Jones)
  - Fixed compilation errors in example code.
- Revision 5, 2015-06-01 (James Jones)
  - Added issues 1 and 2 and made related spec updates.
- Revision 6, 2015-06-01 (James Jones)
  - Merged the platform type mappings table previously removed from VK\_KHR\_swapchain with the platform description table in this spec.
  - Added issues 3 and 4 documenting choices made when building the initial list of native platforms supported.
- Revision 7, 2015-06-11 (Ian Elliott)
  - Updated table 1 per input from the KHR TSG.
  - Updated issue 4 (GBM) per discussion with Daniel Stone. He will create a platform-specific extension sometime in the future.
- Revision 8, 2015-06-17 (James Jones)
  - Updated enum-extending values using new convention.
  - Fixed the value of VK\_SURFACE\_PLATFORM\_INFO\_TYPE\_SUPPORTED\_KHR.
- Revision 9, 2015-06-17 (James Jones)

- Rebased on Vulkan API version 126.
- Revision 10, 2015-06-18 (James Jones)
  - Marked issues 2 and 3 resolved.
- Revision 11, 2015-06-23 (Ian Elliott)
  - Examples now show use of function pointers for extension functions.
  - Eliminated extraneous whitespace.
- Revision 12, 2015-07-07 (Daniel Rakos)
  - Added error section describing when each error is expected to be reported.
  - Replaced the term "queue node index" with "queue family index" in the spec as that is the agreed term to be used in the latest version of the core header and spec.
  - Replaced `bool32_t` with `VkBool32`.
- Revision 13, 2015-08-06 (Daniel Rakos)
  - Updated spec against latest core API header version.
- Revision 14, 2015-08-20 (Ian Elliott)
  - Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  - Switched from "revision" to "version", including use of the `VK_MAKE_VERSION` macro in the header file.
  - Did miscellaneous cleanup, etc.
- Revision 15, 2015-08-20 (Ian Elliott—porting a 2015-07-29 change from James Jones)
  - Moved the surface transform enums here from `VK_WSI_swapchain` so they could be re-used by `VK_WSI_display`.
- Revision 16, 2015-09-01 (James Jones)
  - Restore single-field revision number.
- Revision 17, 2015-09-01 (James Jones)
  - Fix example code compilation errors.
- Revision 18, 2015-09-26 (Jesse Hall)
  - Replaced `VkSurfaceDescriptionKHR` with the `VkSurfaceKHR` object, which is created via layered extensions. Added `VkDestroySurfaceKHR`.
- Revision 19, 2015-09-28 (Jesse Hall)
  - Renamed from `VK_EXT_KHR_swapchain` to `VK_EXT_KHR_surface`.
- Revision 20, 2015-09-30 (Jeff Vigil)
  - Add error result `VK_ERROR_SURFACE_LOST_KHR`.
- Revision 21, 2015-10-15 (Daniel Rakos)
  - Updated the resolution of issue #2 and include the surface capability queries in this extension.

- Renamed SurfaceProperties to SurfaceCapabilities as it better reflects that the values returned are the capabilities of the surface on a particular device.
- Other minor cleanup and consistency changes.
- Revision 22, 2015-10-26 (Ian Elliott)
  - Renamed from VK\_EXT\_KHR\_surface to VK\_KHR\_surface.
- Revision 23, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to vkDestroySurfaceKHR.
- Revision 24, 2015-11-10 (Jesse Hall)
  - Removed VkSurfaceTransformKHR. Use VkSurfaceTransformFlagBitsKHR instead.
  - Rename VkSurfaceCapabilitiesKHR member maxImageArraySize to maxImageArrayLayers.
- Revision 25, 2016-01-14 (James Jones)
  - Moved VK\_ERROR\_NATIVE\_WINDOW\_IN\_USE\_KHR from the VK\_KHR\_android\_surface to the VK\_KHR\_surface extension.
- 2016-08-23 (Ian Elliott)
  - Update the example code, to not have so many characters per line, and to split out a new example to show how to obtain function pointers.
- 2016-08-25 (Ian Elliott)
  - A note was added at the beginning of the example code, stating that it will be removed from future versions of the appendix.

## **VK\_KHR\_surface\_protected\_capabilities**

### **Name String**

`VK_KHR_surface_protected_capabilities`

### **Extension Type**

Instance extension

### **Registered Extension Number**

240

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.1
- Requires `VK_KHR_get_surface_capabilities2`

### **Contact**

- Sandeep Shinde  [@sashinde](#)

### **Last Modified Date**

## IP Status

No known IP claims.

## Contributors

- Sandeep Shinde, NVIDIA
- James Jones, NVIDIA
- Daniel Koch, NVIDIA

This extension extends [VkSurfaceCapabilities2KHR](#), providing applications a way to query whether swapchains **can** be created with the [VK\\_SWAPCHAIN\\_CREATE\\_PROTECTED\\_BIT\\_KHR](#) flag set.

Vulkan 1.1 added (optional) support for protect memory and protected resources including buffers ([VK\\_BUFFER\\_CREATE\\_PROTECTED\\_BIT](#)), images ([VK\\_IMAGE\\_CREATE\\_PROTECTED\\_BIT](#)), and swapchains ([VK\\_SWAPCHAIN\\_CREATE\\_PROTECTED\\_BIT\\_KHR](#)). However, on implementations which support multiple windowing systems, not all window systems **may** be able to provide a protected display path.

This extension provides a way to query if a protected swapchain created for a surface (and thus a specific windowing system) **can** be displayed on screen. It extends the existing [VkSurfaceCapabilities2KHR](#) structure with a new [VkSurfaceProtectedCapabilitiesKHR](#) structure from which the application **can** obtain information about support for protected swapchain creation through [vkGetPhysicalDeviceSurfaceCapabilities2KHR](#).

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_SURFACE\\_PROTECTED\\_CAPABILITIES\\_KHR](#)

## New Enums

None.

## New Structures

- [VkSurfaceProtectedCapabilitiesKHR](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2018-12-18 (Sandeep Shinde, Daniel Koch)
  - Internal revisions.

## VK\_KHR\_swapchain

### Name String

`VK_KHR_swapchain`

### Extension Type

Device extension

### Registered Extension Number

2

### Revision

70

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### Contact

- James Jones [@cubanismo](#)
- Ian Elliott [@ianelliottus](#)

### Last Modified Date

2017-10-06

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Interacts with Vulkan 1.1

### Contributors

- Patrick Doane, Blizzard
- Ian Elliott, LunarG
- Jesse Hall, Google
- Mathias Heyer, NVIDIA
- James Jones, NVIDIA
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung

- Daniel Rakos, AMD
- Graham Sellers, AMD
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG
- Jason Ekstrand, Intel
- Matthaeus G. Chajdas, AMD
- Ray Smith, ARM

The `VK_KHR_swapchain` extension is the device-level companion to the `VK_KHR_surface` extension. It introduces `VkSwapchainKHR` objects, which provide the ability to present rendering results to a surface.

## New Object Types

- `VkSwapchainKHR`

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_PRESENT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR`
  - `VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR`
  - `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`
- Extending `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`
- Extending `VkResult`:
  - `VK_SUBOPTIMAL_KHR`
  - `VK_ERROR_OUT_OF_DATE_KHR`

## New Enums

- `VkDeviceGroupPresentModeFlagBitsKHR`
- `VkSwapchainCreateFlagBitsKHR`

## New Structures

- `VkSwapchainCreateInfoKHR`
- `VkPresentInfoKHR`
- `VkDeviceGroupPresentCapabilitiesKHR`

- [VkImageSwapchainCreateInfoKHR](#)
- [VkBindImageMemorySwapchainCreateInfoKHR](#)
- [VkAcquireNextImageCreateInfoKHR](#)
- [VkDeviceGroupPresentCreateInfoKHR](#)
- [VkDeviceGroupSwapchainCreateInfoKHR](#)

## New Functions

- [vkCreateSwapchainKHR](#)
- [vkDestroySwapchainKHR](#)
- [vkGetSwapchainImagesKHR](#)
- [vkAcquireNextImageKHR](#)
- [vkQueuePresentKHR](#)
- [vkGetDeviceGroupPresentCapabilitiesKHR](#)
- [vkGetDeviceGroupSurfacePresentModesKHR](#)
- [vkGetPhysicalDevicePresentRectanglesKHR](#)
- [vkAcquireNextImage2KHR](#)

## Issues

1) Does this extension allow the application to specify the memory backing of the presentable images?

**RESOLVED:** No. Unlike standard images, the implementation will allocate the memory backing of the presentable image.

2) What operations are allowed on presentable images?

**RESOLVED:** This is determined by the image usage flags specified when creating the presentable image's swapchain.

3) Does this extension support MSAA presentable images?

**RESOLVED:** No. Presentable images are always single-sampled. Multi-sampled rendering must use regular images. To present the rendering results the application must manually resolve the multi-sampled image to a single-sampled presentable image prior to presentation.

4) Does this extension support stereo/multi-view presentable images?

**RESOLVED:** Yes. The number of views associated with a presentable image is determined by the `imageArrayLayers` specified when creating a swapchain. All presentable images in a given swapchain use the same array size.

5) Are the layers of stereo presentable images half-sized?

**RESOLVED:** No. The image extents always match those requested by the application.

6) Do the “present” and “acquire next image” commands operate on a queue? If not, do they need to include explicit semaphore objects to interlock them with queue operations?

**RESOLVED:** The present command operates on a queue. The image ownership operation it represents happens in order with other operations on the queue, so no explicit semaphore object is required to synchronize its actions.

Applications may want to acquire the next image in separate threads from those in which they manage their queue, or in multiple threads. To make such usage easier, the acquire next image command takes a semaphore to signal as a method of explicit synchronization. The application must later queue a wait for this semaphore before queuing execution of any commands using the image.

7) Does [vkAcquireNextImageKHR](#) block if no images are available?

**RESOLVED:** The command takes a timeout parameter. Special values for the timeout are 0, which makes the call a non-blocking operation, and [UINT64\\_MAX](#), which blocks indefinitely. Values in between will block for up to the specified time. The call will return when an image becomes available or an error occurs. It may, but is not required to, return before the specified timeout expires if the swapchain becomes out of date.

8) Can multiple presents be queued using one [vkQueuePresentKHR](#) call?

**RESOLVED:** Yes. [VkPresentInfoKHR](#) contains a list of swapchains and corresponding image indices that will be presented. When supported, all presentations queued with a single [vkQueuePresentKHR](#) call will be applied atomically as one operation. The same swapchain must not appear in the list more than once. Later extensions may provide applications stronger guarantees of atomicity for such present operations, and/or allow them to query whether atomic presentation of a particular group of swapchains is possible.

9) How do the presentation and acquire next image functions notify the application the targeted surface has changed?

**RESOLVED:** Two new result codes are introduced for this purpose:

- [VK\\_SUBOPTIMAL\\_KHR](#) - Presentation will still succeed, subject to the window resize behavior, but the swapchain is no longer configured optimally for the surface it targets. Applications should query updated surface information and recreate their swapchain at the next convenient opportunity.
- [VK\\_ERROR\\_OUT\\_OF\\_DATE\\_KHR](#) - Failure. The swapchain is no longer compatible with the surface it targets. The application must query updated surface information and recreate the swapchain before presentation will succeed.

These can be returned by both [vkAcquireNextImageKHR](#) and [vkQueuePresentKHR](#).

10) Does the [vkAcquireNextImageKHR](#) command return a semaphore to the application via an output parameter, or accept a semaphore to signal from the application as an object handle parameter?

**RESOLVED:** Accept a semaphore to signal as an object handle. This avoids the need to specify whether the application must destroy the semaphore or whether it is owned by the swapchain, and if the latter, what its lifetime is and whether it can be re-used for other operations once it is received from [vkAcquireNextImageKHR](#).

11) What types of swapchain queuing behavior should be exposed? Options include swap interval specification, mailbox/most recent vs. FIFO queue management, targeting specific vertical blank intervals or absolute times for a given present operation, and probably others. For some of these, whether they are specified at swapchain creation time or as per-present parameters needs to be decided as well.

**RESOLVED:** The base swapchain extension will expose 3 possible behaviors (of which, FIFO will always be supported):

- Immediate present: Does not wait for vertical blanking period to update the current image, likely resulting in visible tearing. No internal queue is used. Present requests are applied immediately.
- Mailbox queue: Waits for the next vertical blanking period to update the current image. No tearing should be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for re-use by the application.
- FIFO queue: Waits for the next vertical blanking period to update the current image. No tearing should be observed. An internal queue containing `numSwapchainImages - 1` entries is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty

Not all surfaces will support all of these modes, so the modes supported will be returned using a surface info query. All surfaces must support the FIFO queue mode. Applications must choose one of these modes up front when creating a swapchain. Switching modes can be accomplished by recreating the swapchain.

12) Can `VK_PRESENT_MODE_MAILBOX_KHR` provide non-blocking guarantees for [vkAcquireNextImageKHR](#)? If so, what is the proper criteria?

**RESOLVED:** Yes. The difficulty is not immediately obvious here. Naively, if at least 3 images are requested, mailbox mode should always have an image available for the application if the application does not own any images when the call to [vkAcquireNextImageKHR](#) was made. However, some presentation engines may have more than one “current” image, and would still need to block in some cases. The right requirement appears to be that if the application allocates the surface’s minimum number of images + 1 then it is guaranteed non-blocking behavior when it does not currently own any images.

13) Is there a way to create and initialize a new swapchain for a surface that has generated a `VK_SUBOPTIMAL_KHR` return code while still using the old swapchain?

**RESOLVED:** Not as part of this specification. This could be useful to allow the application to create an “optimal” replacement swapchain and rebuild all its command buffers using it in a background

thread at a low priority while continuing to use the “suboptimal” swapchain in the main thread. It could probably use the same “atomic replace” semantics proposed for recreating direct-to-device swapchains without incurring a mode switch. However, after discussion, it was determined some platforms probably could not support concurrent swapchains for the same surface though, so this will be left out of the base KHR extensions. A future extension could add this for platforms where it is supported.

14) Should there be a special value for `VkSurfaceCapabilitiesKHR::maxImageCount` to indicate there are no practical limits on the number of images in a swapchain?

**RESOLVED:** Yes. There where often be cases where there is no practical limit to the number of images in a swapchain other than the amount of available resources (I.e., memory) in the system. Trying to derive a hard limit from things like memory size is prone to failure. It is better in such cases to leave it to applications to figure such soft limits out via trial/failure iterations.

15) Should there be a special value for `VkSurfaceCapabilitiesKHR::currentExtent` to indicate the size of the platform surface is undefined?

**RESOLVED:** Yes. On some platforms (Wayland, for example), the surface size is defined by the images presented to it rather than the other way around.

16) Should there be a special value for `VkSurfaceCapabilitiesKHR::maxImageExtent` to indicate there is no practical limit on the surface size?

**RESOLVED:** No. It seems unlikely such a system would exist. 0 could be used to indicate the platform places no limits on the extents beyond those imposed by Vulkan for normal images, but this query could just as easily return those same limits, so a special “unlimited” value does not seem useful for this field.

17) How should surface rotation and mirroring be exposed to applications? How do they specify rotation and mirroring transforms applied prior to presentation?

**RESOLVED:** Applications can query both the supported and current transforms of a surface. Both are specified relative to the device’s “natural” display rotation and direction. The supported transforms indicates which orientations the presentation engine accepts images in. For example, a presentation engine that does not support transforming surfaces as part of presentation, and which is presenting to a surface that is displayed with a 90-degree rotation, would return only one supported transform bit: `VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR`. Applications must transform their rendering by the transform they specify when creating the swapchain in `preTransform` field.

18) Can surfaces ever not support `VK_MIRROR_NONE`? Can they support vertical and horizontal mirroring simultaneously? Relatedly, should `VK_MIRROR_NONE[_BIT]` be zero, or bit one, and should applications be allowed to specify multiple pre and current mirror transform bits, or exactly one?

**RESOLVED:** Since some platforms may not support presenting with a transform other than the native window’s current transform, and prerotation/mirroring are specified relative to the device’s natural rotation and direction, rather than relative to the surface’s current rotation and direction, it is necessary to express lack of support for no mirroring. To allow this, the `MIRROR_NONE` enum must occupy a bit in the flags. Since `MIRROR_NONE` must be a bit in the bitmask rather than a bitmask with no values set, allowing more than one bit to be set in the bitmask would make it possible to

describe undefined transforms such as `VK_MIRROR_NONE_BIT` | `VK_MIRROR_HORIZONTAL_BIT`, or a transform that includes both “no mirroring” and “horizontal mirroring” simultaneously. Therefore, it is desirable to allow specifying all supported mirroring transforms using only one bit. The question then becomes, should there be a `VK_MIRROR_HORIZONTAL_AND_VERTICAL_BIT` to represent a simultaneous horizontal and vertical mirror transform? However, such a transform is equivalent to a 180 degree rotation, so presentation engines and applications that wish to support or use such a transform can express it through rotation instead. Therefore, 3 exclusive bits are sufficient to express all needed mirroring transforms.

19) Should support for sRGB be required?

**RESOLVED:** In the advent of UHD and HDR display devices, proper color space information is vital to the display pipeline represented by the swapchain. The app can discover the supported format/color-space pairs and select a pair most suited to its rendering needs. Currently only the sRGB color space is supported, future extensions may provide support for more color spaces. See issues 23 and 24.

20) Is there a mechanism to modify or replace an existing swapchain with one targeting the same surface?

**RESOLVED:** Yes. This is described above in the text.

21) Should there be a way to set prerotation and mirroring using native APIs when presenting using a Vulkan swapchain?

**RESOLVED:** Yes. The transforms that can be expressed in this extension are a subset of those possible on native platforms. If a platform exposes a method to specify the transform of presented images for a given surface using native methods and exposes more transforms or other properties for surfaces than Vulkan supports, it might be impossible, difficult, or inconvenient to set some of those properties using Vulkan KHR extensions and some using the native interfaces. To avoid overwriting properties set using native commands when presenting using a Vulkan swapchain, the application can set the pretransform to “inherit”, in which case the current native properties will be used, or if none are available, a platform-specific default will be used. Platforms that do not specify a reasonable default or do not provide native mechanisms to specify such transforms should not include the inherit bits in the `supportedTransforms` bitmask they return in `VkSurfaceCapabilitiesKHR`.

22) Should the content of presentable images be clipped by objects obscuring their target surface?

**RESOLVED:** Applications can choose which behavior they prefer. Allowing the content to be clipped could enable more optimal presentation methods on some platforms, but some applications might rely on the content of presentable images to perform techniques such as partial updates or motion blurs.

23) What is the purpose of specifying a `VkColorSpaceKHR` along with `VkFormat` when creating a swapchain?

**RESOLVED:** While Vulkan itself is color space agnostic (e.g. even the meaning of R, G, B and A can be freely defined by the rendering application), the swapchain eventually will have to present the images on a display device with specific color reproduction characteristics. If any color space

transformations are necessary before an image can be displayed, the color space of the presented image must be known to the swapchain. A swapchain will only support a restricted set of color format and -space pairs. This set can be discovered via [vkGetPhysicalDeviceSurfaceFormatsKHR](#). As it can be expected that most display devices support the sRGB color space, at least one format/color-space pair has to be exposed, where the color space is [VK\\_COLOR\\_SPACE\\_SRGB\\_NONLINEAR\\_KHR](#).

24) How are sRGB formats and the sRGB color space related?

**RESOLVED:** While Vulkan exposes a number of SRGB texture formats, using such formats does not guarantee working in a specific color space. It merely means that the hardware can directly support applying the non-linear transfer functions defined by the sRGB standard color space when reading from or writing to images of that these formats. Still, it is unlikely that a swapchain will expose a [\\*\\_SRGB](#) format along with any color space other than [VK\\_COLOR\\_SPACE\\_SRGB\\_NONLINEAR\\_KHR](#).

On the other hand, non-[\\*\\_SRGB](#) formats will be very likely exposed in pair with a SRGB color space. This means, the hardware will not apply any transfer function when reading from or writing to such images, yet they will still be presented on a device with sRGB display characteristics. In this case the application is responsible for applying the transfer function, for instance by using shader math.

25) How are the lifetime of surfaces and swapchains targeting them related?

**RESOLVED:** A surface must outlive any swapchains targeting it. A [VkSurfaceKHR](#) owns the binding of the native window to the Vulkan driver.

26) How can the client control the way the alpha channel of swapchain images is treated by the presentation engine during compositing?

**RESOLVED:** We should add new enum values to allow the client to negotiate with the presentation engine on how to treat image alpha values during the compositing process. Since not all platforms can practically control this through the Vulkan driver, a value of [VK\\_COMPOSITE\\_ALPHA\\_INHERIT\\_BIT\\_KHR](#) is provided like for surface transforms.

27) Is [vkCreateSwapchainKHR](#) the right function to return [VK\\_ERROR\\_NATIVE\\_WINDOW\\_IN\\_USE\\_KHR](#), or should the various platform-specific [VkSurfaceKHR](#) factory functions catch this error earlier?

**RESOLVED:** For most platforms, the [VkSurfaceKHR](#) structure is a simple container holding the data that identifies a native window or other object representing a surface on a particular platform. For the surface factory functions to return this error, they would likely need to register a reference on the native objects with the native display server somehow, and ensure no other such references exist. Surfaces were not intended to be that heavyweight.

Swapchains are intended to be the objects that directly manipulate native windows and communicate with the native presentation mechanisms. Swapchains will already need to communicate with the native display server to negotiate allocation and/or presentation of presentable images for a native surface. Therefore, it makes more sense for swapchain creation to be the point at which native object exclusivity is enforced. Platforms may choose to enforce further restrictions on the number of [VkSurfaceKHR](#) objects that may be created for the same native window if such a requirement makes sense on a particular platform, but a global requirement is only sensible at the swapchain level.

## Examples

### Note

The example code for the `VK_KHR_surface` and `VK_KHR_swapchain` extensions was removed from the appendix after revision 1.0.29. This WSI example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: <https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c>).



## Version History

- Revision 1, 2015-05-20 (James Jones)
  - Initial draft, based on LunarG KHR spec, other KHR specs, patches attached to bugs.
- Revision 2, 2015-05-22 (Ian Elliott)
  - Made many agreed-upon changes from 2015-05-21 KHR TSG meeting. This includes using only a queue for presentation, and having an explicit function to acquire the next image.
  - Fixed typos and other minor mistakes.
- Revision 3, 2015-05-26 (Ian Elliott)
  - Improved the Description section.
  - Added or resolved issues that were found in improving the Description. For example, `pSurfaceDescription` is used consistently, instead of sometimes using `pSurface`.
- Revision 4, 2015-05-27 (James Jones)
  - Fixed some grammatical errors and typos
  - Filled in the description of `imageUseFlags` when creating a swapchain.
  - Added a description of `swapInterval`.
  - Replaced the paragraph describing the order of operations on a queue for image ownership and presentation.
- Revision 5, 2015-05-27 (James Jones)
  - Imported relevant issues from the (abandoned) `vk_wsi_persistent_swapchain_images` extension.
  - Added issues 6 and 7, regarding behavior of the acquire next image and present commands with respect to queues.
  - Updated spec language and examples to align with proposed resolutions to issues 6 and 7.
- Revision 6, 2015-05-27 (James Jones)
  - Added issue 8, regarding atomic presentation of multiple swapchains
  - Updated spec language and examples to align with proposed resolution to issue 8.
- Revision 7, 2015-05-27 (James Jones)
  - Fixed compilation errors in example code, and made related spec fixes.
- Revision 8, 2015-05-27 (James Jones)

- Added issue 9, and the related VK\_SUBOPTIMAL\_KHR result code.
- Renamed VK\_OUT\_OF\_DATE\_KHR to VK\_ERROR\_OUT\_OF\_DATE\_KHR.
- Revision 9, 2015-05-27 (James Jones)
  - Added inline proposed resolutions (marked with [JRJ]) to some XXX questions/issues. These should be moved to the issues section in a subsequent update if the proposals are adopted.
- Revision 10, 2015-05-28 (James Jones)
  - Converted vkAcquireNextImageKHR back to a non-queue operation that uses a VkSemaphore object for explicit synchronization.
  - Added issue 10 to determine whether vkAcquireNextImageKHR generates or returns semaphores, or whether it operates on a semaphore provided by the application.
- Revision 11, 2015-05-28 (James Jones)
  - Marked issues 6, 7, and 8 resolved.
  - Renamed VkSurfaceCapabilityPropertiesKHR to VkSurfacePropertiesKHR to better convey the mutable nature of the info it contains.
- Revision 12, 2015-05-28 (James Jones)
  - Added issue 11 with a proposed resolution, and the related issue 12.
  - Updated various sections of the spec to match the proposed resolution to issue 11.
- Revision 13, 2015-06-01 (James Jones)
  - Moved some structures to VK\_EXT\_KHR\_swap\_chain to resolve the spec's issues 1 and 2.
- Revision 14, 2015-06-01 (James Jones)
  - Added code for example 4 demonstrating how an application might make use of the two different present and acquire next image KHR result codes.
  - Added issue 13.
- Revision 15, 2015-06-01 (James Jones)
  - Added issues 14 - 16 and related spec language.
  - Fixed some spelling errors.
  - Added language describing the meaningful return values for vkAcquireNextImageKHR and vkQueuePresentKHR.
- Revision 16, 2015-06-02 (James Jones)
  - Added issues 17 and 18, as well as related spec language.
  - Removed some erroneous text added by mistake in the last update.
- Revision 17, 2015-06-15 (Ian Elliott)
  - Changed special value from "-1" to "0" so that the data types can be unsigned.
- Revision 18, 2015-06-15 (Ian Elliott)
  - Clarified the values of VkSurfacePropertiesKHR::minImageCount and the timeout parameter of the vkAcquireNextImageKHR function.
- Revision 19, 2015-06-17 (James Jones)

- Misc. cleanup. Removed resolved inline issues and fixed typos.
- Fixed clarification of VkSurfacePropertiesKHR::minImageCount made in version 18.
- Added a brief "Image Ownership" definition to the list of terms used in the spec.
- Revision 20, 2015-06-17 (James Jones)
  - Updated enum-extending values using new convention.
- Revision 21, 2015-06-17 (James Jones)
  - Added language describing how to use VK\_IMAGE\_LAYOUT\_PRESENT\_SOURCE\_KHR.
  - Cleaned up an XXX comment regarding the description of which queues vkQueuePresentKHR can be used on.
- Revision 22, 2015-06-17 (James Jones)
  - Rebased on Vulkan API version 126.
- Revision 23, 2015-06-18 (James Jones)
  - Updated language for issue 12 to read as a proposed resolution.
  - Marked issues 11, 12, 13, 16, and 17 resolved.
  - Temporarily added links to the relevant bugs under the remaining unresolved issues.
  - Added issues 19 and 20 as well as proposed resolutions.
- Revision 24, 2015-06-19 (Ian Elliott)
  - Changed special value for VkSurfacePropertiesKHR::currentExtent back to "-1" from "0". This value will never need to be unsigned, and "0" is actually a legal value.
- Revision 25, 2015-06-23 (Ian Elliott)
  - Examples now show use of function pointers for extension functions.
  - Eliminated extraneous whitespace.
- Revision 26, 2015-06-25 (Ian Elliott)
  - Resolved Issues 9 & 10 per KHR TSG meeting.
- Revision 27, 2015-06-25 (James Jones)
  - Added oldSwapchain member to VkSwapchainCreateInfoKHR.
- Revision 28, 2015-06-25 (James Jones)
  - Added the "inherit" bits to the rotation and mirroring flags and the associated issue 21.
- Revision 29, 2015-06-25 (James Jones)
  - Added the "clipped" flag to VkSwapchainCreateInfoKHR, and the associated issue 22.
  - Specified that presenting an image does not modify it.
- Revision 30, 2015-06-25 (James Jones)
  - Added language to the spec that clarifies the behavior of vkCreateSwapchainKHR() when the oldSwapchain field of VkSwapchainCreateInfoKHR is not NULL.
- Revision 31, 2015-06-26 (Ian Elliott)

- Example of new VkSwapchainCreateInfoKHR members, "oldSwapchain" and "clipped".
- Example of using VkSurfacePropertiesKHR::{min|max}ImageCount to set VkSwapchainCreateInfoKHR::minImageCount.
- Rename vkGetSurfaceInfoKHR()'s 4th parameter to "pDataSize", for consistency with other functions.
- Add macro with C-string name of extension (just to header file).
- Revision 32, 2015-06-26 (James Jones)
  - Minor adjustments to the language describing the behavior of "oldSwapchain"
  - Fixed the version date on my previous two updates.
- Revision 33, 2015-06-26 (Jesse Hall)
  - Add usage flags to VkSwapchainCreateInfoKHR
- Revision 34, 2015-06-26 (Ian Elliott)
  - Rename vkQueuePresentKHR()'s 2nd parameter to "pPresentInfo", for consistency with other functions.
- Revision 35, 2015-06-26 (Jason Ekstrand)
  - Merged the VkRotationFlagBitsKHR and VkMirrorFlagBitsKHR enums into a single VkSurfaceTransformFlagBitsKHR enum.
- Revision 36, 2015-06-26 (Jason Ekstrand)
  - Added a VkSurfaceTransformKHR enum that is not a bitmask. Each value in VkSurfaceTransformKHR corresponds directly to one of the bits in VkSurfaceTransformFlagBitsKHR so transforming from one to the other is easy. Having a separate enum means that currentTransform and preTransform are now unambiguous by definition.
- Revision 37, 2015-06-29 (Ian Elliott)
  - Corrected one of the signatures of vkAcquireNextImageKHR, which had the last two parameters switched from what it is elsewhere in the specification and header files.
- Revision 38, 2015-06-30 (Ian Elliott)
  - Corrected a typo in description of the vkGetSwapchainInfoKHR() function.
  - Corrected a typo in header file comment for VkPresentInfoKHR::sType.
- Revision 39, 2015-07-07 (Daniel Rakos)
  - Added error section describing when each error is expected to be reported.
  - Replaced bool32\_t with VkBool32.
- Revision 40, 2015-07-10 (Ian Elliott)
  - Updated to work with version 138 of the "vulkan.h" header. This includes declaring the VkSwapchainKHR type using the new VK\_DEFINE\_NONDISP\_HANDLE macro, and no longer extending VkObjectType (which was eliminated).
- Revision 41 2015-07-09 (Mathias Heyer)
  - Added color space language.

- Revision 42, 2015-07-10 (Daniel Rakos)
  - Updated query mechanism to reflect the convention changes done in the core spec.
  - Removed "queue" from the name of VK\_STRUCTURE\_TYPE\_QUEUE\_PRESENT\_INFO\_KHR to be consistent with the established naming convention.
  - Removed reference to the no longer existing VkObjectType enum.
- Revision 43, 2015-07-17 (Daniel Rakos)
  - Added support for concurrent sharing of swapchain images across queue families.
  - Updated sample code based on recent changes
- Revision 44, 2015-07-27 (Ian Elliott)
  - Noted that support for VK\_PRESENT\_MODE\_FIFO\_KHR is required. That is ICDs may optionally support IMMEDIATE and MAILBOX, but must support FIFO.
- Revision 45, 2015-08-07 (Ian Elliott)
  - Corrected a typo in spec file (type and variable name had wrong case for the imageColorSpace member of the VkSwapchainCreateInfoKHR struct).
  - Corrected a typo in header file (last parameter in PFN\_vkGetSurfacePropertiesKHR was missing "KHR" at the end of type: VkSurfacePropertiesKHR).
- Revision 46, 2015-08-20 (Ian Elliott)
  - Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  - Switched from "revision" to "version", including use of the VK\_MAKE\_VERSION macro in the header file.
  - Made improvements to several descriptions.
  - Changed the status of several issues from PROPOSED to RESOLVED, leaving no unresolved issues.
  - Resolved several TODOs, did miscellaneous cleanup, etc.
- Revision 47, 2015-08-20 (Ian Elliott—porting a 2015-07-29 change from James Jones)
  - Moved the surface transform enums to VK\_WSI\_swapchain so they could be re-used by VK\_WSI\_display.
- Revision 48, 2015-09-01 (James Jones)
  - Various minor cleanups.
- Revision 49, 2015-09-01 (James Jones)
  - Restore single-field revision number.
- Revision 50, 2015-09-01 (James Jones)
  - Update Example #4 to include code that illustrates how to use the oldSwapchain field.
- Revision 51, 2015-09-01 (James Jones)
  - Fix example code compilation errors.
- Revision 52, 2015-09-08 (Matthaeus G. Chajdas)

- Corrected a typo.
- Revision 53, 2015-09-10 (Alon Or-bach)
  - Removed underscore from SWAP\_CHAIN left in VK\_STRUCTURE\_TYPE\_SWAPCHAIN\_CREATE\_INFO\_KHR.
- Revision 54, 2015-09-11 (Jesse Hall)
  - Described the execution and memory coherence requirements for image transitions to and from VK\_IMAGE\_LAYOUT\_PRESENT\_SOURCE\_KHR.
- Revision 55, 2015-09-11 (Ray Smith)
  - Added errors for destroying and binding memory to presentable images
- Revision 56, 2015-09-18 (James Jones)
  - Added fence argument to vkAcquireNextImageKHR
  - Added example of how to meter a host thread based on presentation rate.
- Revision 57, 2015-09-26 (Jesse Hall)
  - Replace VkSurfaceDescriptionKHR with VkSurfaceKHR.
  - Added issue 25 with agreed resolution.
- Revision 58, 2015-09-28 (Jesse Hall)
  - Renamed from VK\_EXT\_KHR\_device\_swapchain to VK\_EXT\_KHR\_swapchain.
- Revision 59, 2015-09-29 (Ian Elliott)
  - Changed vkDestroySwapchainKHR() to return void.
- Revision 60, 2015-10-01 (Jeff Vigil)
  - Added error result VK\_ERROR\_SURFACE\_LOST\_KHR.
- Revision 61, 2015-10-05 (Jason Ekstrand)
  - Added the VkCompositeAlpha enum and corresponding structure fields.
- Revision 62, 2015-10-12 (Daniel Rakos)
  - Added VK\_PRESENT\_MODE\_FIFO\_RELAXED\_KHR.
- Revision 63, 2015-10-15 (Daniel Rakos)
  - Moved surface capability queries to VK\_EXT\_KHR\_surface.
- Revision 64, 2015-10-26 (Ian Elliott)
  - Renamed from VK\_EXT\_KHR\_swapchain to VK\_KHR\_swapchain.
- Revision 65, 2015-10-28 (Ian Elliott)
  - Added optional pResult member to VkPresentInfoKHR, so that per-swapchain results can be obtained from vkQueuePresentKHR().
- Revision 66, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to create and destroy functions.
  - Updated resource transition language.

- Updated sample code.
- Revision 67, 2015-11-10 (Jesse Hall)
  - Add reserved flags bitmask to VkSwapchainCreateInfoKHR.
  - Modify naming and member ordering to match API style conventions, and so the VkSwapchainCreateInfoKHR image property members mirror corresponding VkImageCreateInfo members but with an 'image' prefix.
  - Make VkPresentInfoKHR::pResults non-const; it is an output array parameter.
  - Make pPresentInfo parameter to vkQueuePresentKHR const.
- Revision 68, 2016-04-05 (Ian Elliott)
  - Moved the "validity" include for vkAcquireNextImage to be in its proper place, after the prototype and list of parameters.
  - Clarified language about presentable images, including how they are acquired, when applications can and cannot use them, etc. As part of this, removed language about "ownership" of presentable images, and replaced it with more-consistent language about presentable images being "acquired" by the application.
- 2016-08-23 (Ian Elliott)
  - Update the example code, to use the final API command names, to not have so many characters per line, and to split out a new example to show how to obtain function pointers. This code is more similar to the LunarG "cube" demo program.
- 2016-08-25 (Ian Elliott)
  - A note was added at the beginning of the example code, stating that it will be removed from future versions of the appendix.
- Revision 69, 2017-09-07 (Tobias Hector)
  - Added interactions with Vulkan 1.1
- Revision 70, 2017-10-06 (Ian Elliott)
  - Corrected interactions with Vulkan 1.1

## **VK\_KHR\_swapchain mutable\_format**

### **Name String**

**VK\_KHR\_swapchain mutable\_format**

### **Extension Type**

Device extension

### **Registered Extension Number**

201

### **Revision**

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_swapchain`
- Requires `VK_KHR_maintenance2`
- Requires `VK_KHR_image_format_list`

## Contact

- Daniel Rakos [@drakos-arm](#)

## Last Modified Date

2018-03-28

## IP Status

No known IP claims.

## Contributors

- Jason Ekstrand, Intel
- Jan-Harald Fredriksen, ARM
- Jesse Hall, Google
- Daniel Rakos, AMD
- Ray Smith, ARM

## Short Description

Allows processing of swapchain images as different formats to that used by the window system, which is particularly useful for switching between sRGB and linear RGB formats.

## Description

This extension adds a new swapchain creation flag that enables creating image views from presentable images with a different format than the one used to create the swapchain.

## New Object Types

None.

## New Enum Constants

- Extending `VkSwapchainCreateFlagBitsKHR`:
  - `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR`

## New Enums

None.

## New Structures

None.

## New Functions

None.

## Issues

1) Are there any new capabilities needed?

**RESOLVED:** No. It is expected that all implementations exposing this extension support swapchain image format mutability.

2) Do we need a separate `VK_SWAPCHAIN_CREATE_EXTENDED_USAGE_BIT_KHR`?

**RESOLVED:** No. This extension requires `VK_KHR_maintenance2` and presentable images of swapchains created with `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR` are created internally in a way equivalent to specifying both `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` and `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR`.

3) Do we need a separate structure to allow specifying an image format list for swapchains?

**RESOLVED:** No. We simply use the same `VkImageFormatListCreateInfoKHR` structure introduced by `VK_KHR_image_format_list`. The structure is required to be included in the `pNext` chain of `VkSwapchainCreateInfoKHR` for swapchains created with `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR`.

## Version History

- Revision 1, 2018-03-28 (Daniel Rakos)
  - Internal revisions.

## `VK_KHR_timeline_semaphore`

### Name String

`VK_KHR_timeline_semaphore`

### Extension Type

Device extension

### Registered Extension Number

208

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0

- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Jason Ekstrand [@jekstrand](#)

## Last Modified Date

2019-06-12

## IP Status

No known IP claims.

## Interactions and External Dependencies

- This extension interacts with [VK\\_KHR\\_external\\_semaphore\\_capabilities](#)
- This extension interacts with [VK\\_KHR\\_external\\_semaphore](#)
- This extension interacts with [VK\\_KHR\\_external\\_semaphore\\_win32](#)

## Contributors

- Jeff Bolz, NVIDIA
- Yuriy O'Donnell, Epic Games
- Jason Ekstrand, Intel
- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Daniel Rakos, AMD
- Ray Smith, Arm

This extension introduces a new type of semaphore that has an integer payload identifying a point in a timeline. Such timeline semaphores support the following operations:

- Host query - A host operation that allows querying the payload of the timeline semaphore.
- Host wait - A host operation that allows a blocking wait for a timeline semaphore to reach a specified value.
- Host signal - A host operation that allows advancing the timeline semaphore to a specified value.
- Device wait - A device operation that allows waiting for a timeline semaphore to reach a specified value.
- Device signal - A device operation that allows advancing the timeline semaphore to a specified value.

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES_KHR`
  - `VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO_KHR`

## New Enums

- [VkSemaphoreTypeKHR](#)

## New Structs

- [VkPhysicalDeviceTimelineSemaphoreFeaturesKHR](#)
- [VkPhysicalDeviceTimelineSemaphorePropertiesKHR](#)
- [VkSemaphoreTypeCreateInfoKHR](#)
- [VkTimelineSemaphoreSubmitInfoKHR](#)
- [VkSemaphoreWaitInfoKHR](#)
- [VkSemaphoreSignalInfoKHR](#)

## New Functions

- [vkGetSemaphoreCounterValueKHR](#)
- [vkWaitSemaphoresKHR](#)
- [vkSignalSemaphoreKHR](#)

## Issues

1) Do we need a new object type for this?

**RESOLVED:** No, we just introduce a new type of semaphore object, as `VK_KHR_external_semaphore_win32` already uses semaphores as the destination for importing D3D12 fence objects, which are semantically close/identical to the proposed synchronization primitive.

2) What type of payload the new synchronization primitive has?

**RESOLVED:** A 64-bit unsigned integer that can only be set to monotonically increasing values by signal operations and is not changed by wait operations.

3) Does the new synchronization primitive have the same signal-before-wait requirement as the existing semaphores do?

**RESOLVED:** No. Timeline semaphores support signaling and waiting entirely asynchronously. It is

the responsibility of the client to avoid deadlock.

4) Does the new synchronization primitive allow resetting its payload?

**RESOLVED:** No, allowing the payload value to "go backwards" is problematic. Applications looking for reset behavior should create a new instance of the synchronization primitive instead.

5) How do we enable host waits on the synchronization primitive?

**RESOLVED:** Both a non-blocking query of the current payload value of the synchronization primitive, and a blocking wait operation are provided.

6) How do we enable device waits and signals on the synchronization primitive?

**RESOLVED:** Similar to `VK_KHR_external_semaphore_win32`, this extension introduces a new structure that can be chained to `VkSubmitInfo` to specify the values signaled semaphores should be set to, and the values waited semaphores need to reach.

7) Can the new synchronization primitive be used to synchronize presentation and swapchain image acquisition operations?

**RESOLVED:** Some implementations may have problems with supporting that directly, thus it's not allowed in this extension.

8) Do we want to support external sharing of the new synchronization primitive type?

**RESOLVED:** Yes. Timeline semaphore specific external sharing capabilities can be queried using `vkGetPhysicalDeviceExternalSemaphoreProperties` by chaining the new `VkSemaphoreTypeCreateInfoKHR` structure to its `pExternalSemaphoreInfo` structure. This allows having a different set of external semaphore handle types supported for timeline semaphores vs binary semaphores.

9) Do we need to add a host signal operation for the new synchronization primitive type?

**RESOLVED:** Yes. This helps in situations where one host thread submits a workload but another host thread has the information on when the workload is ready to be executed.

10) How should the new synchronization primitive interact with the ordering requirements of the original `VkSemaphore`?

**RESOLVED:** Prior to calling any command which **may** cause a wait operation on a binary semaphore, the client **must** ensure that the semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) **must** have also been submitted for execution.

11) Should we have separate feature bits for different sub-features of timeline semaphores?

**RESOLVED:** No. The only feature which cannot be supported universally is timeline semaphore import/export. For import/export, the client is already required to query available external handle types via `vkGetPhysicalDeviceExternalSemaphoreProperties` and provide the semaphore type by adding an instance of `VkSemaphoreTypeCreateInfoKHR` to the `pNext` chain of

[VkPhysicalDeviceExternalSemaphoreInfo](#) so no new feature bit is required.

## Version History

- Revision 1, 2018-05-10 (Jason Ekstrand)
  - Initial version
- Revision 2, 2019-06-12 (Jason Ekstrand)
  - Added an initialValue parameter to timeline semaphore creation

## VK\_KHR\_uniform\_buffer\_standard\_layout

### Name String

`VK_KHR_uniform_buffer_standard_layout`

### Extension Type

Device extension

### Registered Extension Number

254

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Graeme Leese [@gnl21](#)

### Last Modified Date

2019-01-25

### Contributors

- Graeme Leese, Broadcom
- Jeff Bolz, NVIDIA
- Tobias Hector, AMD
- Jason Ekstrand, Intel
- Neil Henning, AMD

## Short Description

Enables tighter array and struct packing to be used with uniform buffers.

## Description

This extension modifies the alignment rules for uniform buffers, allowing for tighter packing of arrays and structures. This allows, for example, the std430 layout, as defined in [GLSL](#) to be supported in uniform buffers.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES_KHR`

## New Structures

- [VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR](#)

## Issues

None.

## Version History

- Revision 1, 2019-01-25 (Graeme Leese)
  - Initial draft

## `VK_KHR_vulkan_memory_model`

### Name String

`VK_KHR_vulkan_memory_model`

### Extension Type

Device extension

### Registered Extension Number

212

### Revision

3

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Jeff Bolz [@jeffbolznv](#)

### Last Modified Date

2018-12-10

### IP Status

No known IP claims.

## Interactions and External Dependencies

- This extension requires [SPV\\_KHR\\_vulkan\\_memory\\_model](#)

## Contributors

- Jeff Bolz, NVIDIA
- Alan Baker, Google
- Tobias Hector, AMD
- David Neto, Google
- Robert Simpson, Qualcomm Technologies, Inc.
- Brian Sumner, AMD

The [VK\\_KHR\\_vulkan\\_memory\\_model](#) extension allows use of the [Vulkan Memory Model](#), which formally defines how to synchronize memory accesses to the same memory locations performed by multiple shader invocations.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_VULKAN\\_MEMORY\\_MODEL\\_FEATURES\\_KHR](#)

## New Structures

- [VkPhysicalDeviceVulkanMemoryModelFeaturesKHR](#)

*Note*



Version 3 of the spec added a member ([vulkanMemoryModelAvailabilityVisibilityChains](#)) to [VkPhysicalDeviceVulkanMemoryModelFeaturesKHR](#), which is an incompatible change from version 2.

## New SPIR-V Capabilities

- [VulkanMemoryModelKHR](#)

## Issues

## Version History

- Revision 1, 2018-06-24 (Jeff Bolz)
  - Initial draft
- Revision 3, 2018-12-10 (Jeff Bolz)
  - Add `vulkanMemoryModelAvailabilityVisibilityChains` member to the `VkPhysicalDeviceVulkanMemoryModelFeaturesKHR` structure.

## **VK\_KHR\_wayland\_surface**

### **Name String**

`VK_KHR_wayland_surface`

### **Extension Type**

Instance extension

### **Registered Extension Number**

7

### **Revision**

6

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### **Contact**

- Jesse Hall [@critsec](#)
- Ian Elliott [@ianelliottus](#)

### **Last Modified Date**

2015-11-28

### **IP Status**

No known IP claims.

### **Contributors**

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel
- Ian Elliott, LunarG
- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD

- Ray Smith, ARM
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG

The `VK_KHR_wayland_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to a Wayland `wl_surface`, as well as a query to determine support for rendering to a Wayland compositor.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_WAYLAND_SURFACE_CREATE_INFO_KHR`

## New Enums

None

## New Structures

- `VkWaylandSurfaceCreateInfoKHR`

## New Functions

- `vkCreateWaylandSurfaceKHR`
- `vkGetPhysicalDeviceWaylandPresentationSupportKHR`

## Issues

1) Does Wayland need a way to query for compatibility between a particular physical device and a specific Wayland display? This would be a more general query than `vkGetPhysicalDeviceSurfaceSupportKHR`: if the Wayland-specific query returned `VK_TRUE` for a (`VkPhysicalDevice`, `struct wl_display*`) pair, then the physical device could be assumed to support presentation to any `VkSurfaceKHR` for surfaces on the display.

**RESOLVED:** Yes. `vkGetPhysicalDeviceWaylandPresentationSupportKHR` was added to address this issue.

2) Should we require surfaces created with `vkCreateWaylandSurfaceKHR` to support the `VK_PRESENT_MODE_MAILBOX_KHR` present mode?

**RESOLVED:** Yes. Wayland is an inherently mailbox window system and mailbox support is required for some Wayland compositor interactions to work as expected. While handling these interactions may be possible with `VK_PRESENT_MODE_FIFO_KHR`, it is much more difficult to do without deadlock and requiring all Wayland applications to be able to support implementations which only

support `VK_PRESENT_MODE_FIFO_KHR` would be an onerous restriction on application developers.

## Version History

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft, based on the previous contents of `VK_EXT_KHR_swapchain` (later renamed `VK_EXT_KHR_surface`).
- Revision 2, 2015-10-02 (James Jones)
  - Added `vkGetPhysicalDeviceWaylandPresentationSupportKHR()` to resolve issue #1.
  - Adjusted wording of issue #1 to match the agreed-upon solution.
  - Renamed "window" parameters to "surface" to match Wayland conventions.
- Revision 3, 2015-10-26 (Ian Elliott)
  - Renamed from `VK_EXT_KHR_wayland_surface` to `VK_KHR_wayland_surface`.
- Revision 4, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to `vkCreateWaylandSurfaceKHR`.
- Revision 5, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a `pCreateInfo` structure.
- Revision 6, 2017-02-08 (Jason Ekstrand)
  - Added the requirement that implementations support `VK_PRESENT_MODE_MAILBOX_KHR`.
  - Added wording about interactions between `vkQueuePresentKHR` and the Wayland requests sent to the compositor.

## `VK_KHR_win32_keyed_mutex`

### Name String

`VK_KHR_win32_keyed_mutex`

### Extension Type

Device extension

### Registered Extension Number

76

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_external_memory_win32`

### Contact

- Carsten Rohde [Ocrohde](#)

## Last Modified Date

2016-10-21

## IP Status

No known IP claims.

## Contributors

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

Applications that wish to import Direct3D 11 memory objects into the Vulkan API may wish to use the native keyed mutex mechanism to synchronize access to the memory between Vulkan and Direct3D. This extension provides a way for an application to access the keyed mutex associated with an imported Vulkan memory object when submitting command buffers to a queue.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHR`

## New Enums

None.

## New Structs

- `VkWin32KeyedMutexAcquireReleaseInfoKHR`

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

## `VK_KHR_win32_surface`

**Name String**

`VK_KHR_win32_surface`

**Extension Type**

Instance extension

**Registered Extension Number**

10

**Revision**

6

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

**Contact**

- Jesse Hall [critsec](#)
- Ian Elliott [ianelliottus](#)

**Last Modified Date**

2017-04-24

**IP Status**

No known IP claims.

**Contributors**

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel
- Ian Elliott, LunarG
- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Ray Smith, ARM
- Jeff Vigil, Qualcomm

- Chia-I Wu, LunarG

The `VK_KHR_win32_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to a Win32 `HWND`, as well as a query to determine support for rendering to the windows desktop.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR`

## New Enums

None

## New Structures

- `VkWin32SurfaceCreateInfoKHR`

## New Functions

- `vkCreateWin32SurfaceKHR`
- `vkGetPhysicalDeviceWin32PresentationSupportKHR`

## Issues

1) Does Win32 need a way to query for compatibility between a particular physical device and a specific screen? Compatibility between a physical device and a window generally only depends on what screen the window is on. However, there is not an obvious way to identify a screen without already having a window on the screen.

**RESOLVED:** No. While it may be useful, there is not a clear way to do this on Win32. However, a method was added to query support for presenting to the windows desktop as a whole.

2) If a native window object (`HWND`) is used by one graphics API, and then is later used by a different graphics API (one of which is Vulkan), can these uses interfere with each other?

**RESOLVED:** Yes.

Uses of a window object by multiple graphics APIs results in undefined behavior. Such behavior may succeed when using one Vulkan implementation but fail when using a different Vulkan implementation. Potential failures include:

- Creating then destroying a flip presentation model DXGI swapchain on a window object can prevent `vkCreateSwapchainKHR` from succeeding on the same window object.

- Creating then destroying a [VkSwapchainKHR](#) on a window object can prevent creation of a bitblt model DXGI swapchain on the same window object.
- Creating then destroying a [VkSwapchainKHR](#) on a window object can effectively [SetPixelFormat](#) to a different format than the format chosen by an OpenGL application.
- Creating then destroying a [VkSwapchainKHR](#) on a window object on one [VkPhysicalDevice](#) can prevent [vkCreateSwapchainKHR](#) from succeeding on the same window object, but on a different [VkPhysicalDevice](#) that is associated with a different Vulkan ICD.

In all cases the problem can be worked around by creating a new window object.

Technical details include:

- Creating a DXGI swapchain over a window object can alter the object for the remainder of its lifetime. The alteration persists even after the DXGI swapchain has been destroyed. This alteration can make it impossible for a conformant Vulkan implementation to create a [VkSwapchainKHR](#) over the same window object. Mention of this alteration can be found in the remarks section of the MSDN documentation for [DXGI\\_SWAP\\_EFFECT](#).
- Calling GDI's [SetPixelFormat](#) (needed by OpenGL's WGL layer) on a window object alters the object for the remainder of its lifetime. The MSDN documentation for [SetPixelFormat](#) explains that a window object's pixel format can be set only one time.
- Creating a [VkSwapchainKHR](#) over a window object can alter the object for the remaining life of its lifetime. Either of the above alterations may occur as a side-effect of [VkSwapchainKHR](#).

## Version History

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft, based on the previous contents of VK\_EXT\_KHR\_swapchain (later renamed VK\_EXT\_KHR\_surface).
- Revision 2, 2015-10-02 (James Jones)
  - Added presentation support query for win32 desktops.
- Revision 3, 2015-10-26 (Ian Elliott)
  - Renamed from VK\_EXT\_KHR\_win32\_surface to VK\_KHR\_win32\_surface.
- Revision 4, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to [vkCreateWin32SurfaceKHR](#).
- Revision 5, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a [pCreateInfo](#) structure.
- Revision 6, 2017-04-24 (Jeff Juliano)
  - Add issue 2 addressing reuse of a native window object in a different Graphics API, or by a different Vulkan ICD.

## VK\_KHR\_xcb\_surface

**Name String**

`VK_KHR_xcb_surface`

**Extension Type**

Instance extension

**Registered Extension Number**

6

**Revision**

6

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

**Contact**

- Jesse Hall [@critsec](#)
- Ian Elliott [@ianelliottus](#)

**Last Modified Date**

2015-11-28

**IP Status**

No known IP claims.

**Contributors**

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel
- Ian Elliott, LunarG
- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Ray Smith, ARM
- Jeff Vigil, Qualcomm

- Chia-I Wu, LunarG

The `VK_KHR_xcb_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to an X11 `Window`, using the XCB client-side library, as well as a query to determine support for rendering via XCB.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR`

## New Enums

None

## New Structures

- `VkXcbSurfaceCreateInfoKHR`

## New Functions

- `vkCreateXcbSurfaceKHR`
- `vkGetPhysicalDeviceXcbPresentationSupportKHR`

## Issues

1) Does XCB need a way to query for compatibility between a particular physical device and a specific screen? This would be a more general query than `vkGetPhysicalDeviceSurfaceSupportKHR`: If it returned `VK_TRUE`, then the physical device could be assumed to support presentation to any window on that screen.

**RESOLVED:** Yes, this is needed for toolkits that want to create a `VkDevice` before creating a window. To ensure the query is reliable, it must be made against a particular X visual rather than the screen in general.

## Version History

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft, based on the previous contents of `VK_EXT_KHR_swapchain` (later renamed `VK_EXT_KHR_surface`).
- Revision 2, 2015-10-02 (James Jones)
  - Added presentation support query for an (`xcb_connection_t*`, `xcb_visualid_t`) pair.
  - Removed "root" parameter from `CreateXcbSurfaceKHR()`, as it is redundant when a window

on the same screen is specified as well.

- Adjusted wording of issue #1 and added agreed upon resolution.
- Revision 3, 2015-10-14 (Ian Elliott)
  - Removed "root" parameter from CreateXcbSurfaceKHR() in one more place.
- Revision 4, 2015-10-26 (Ian Elliott)
  - Renamed from VK\_EXT\_KHR\_xcb\_surface to VK\_KHR\_xcb\_surface.
- Revision 5, 2015-10-23 (Daniel Rakos)
  - Added allocation callbacks to vkCreateXcbSurfaceKHR.
- Revision 6, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a pCreateInfo structure.

## **VK\_KHR\_xlib\_surface**

### **Name String**

[VK\\_KHR\\_xlib\\_surface](#)

### **Extension Type**

Instance extension

### **Registered Extension Number**

5

### **Revision**

6

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### **Contact**

- Jesse Hall [@critsec](#)
- Ian Elliott [@ianelliottus](#)

### **Last Modified Date**

2015-11-28

### **IP Status**

No known IP claims.

### **Contributors**

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel
- Ian Elliott, LunarG

- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Ray Smith, ARM
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG

The `VK_KHR_xlib_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to an X11 `Window`, using the Xlib client-side library, as well as a query to determine support for rendering via Xlib.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR`

## New Enums

None

## New Structures

- `VkXlibSurfaceCreateInfoKHR`

## New Functions

- `vkCreateXlibSurfaceKHR`
- `vkGetPhysicalDeviceXlibPresentationSupportKHR`

## Issues

1) Does X11 need a way to query for compatibility between a particular physical device and a specific screen? This would be a more general query than `vkGetPhysicalDeviceSurfaceSupportKHR`;

if it returned `VK_TRUE`, then the physical device could be assumed to support presentation to any window on that screen.

**RESOLVED:** Yes, this is needed for toolkits that want to create a `VkDevice` before creating a window. To ensure the query is reliable, it must be made against a particular X visual rather than the screen in general.

## Version History

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft, based on the previous contents of `VK_EXT_KHR_swapchain` (later renamed `VK_EXT_KHR_surface`).
- Revision 2, 2015-10-02 (James Jones)
  - Added presentation support query for (`Display*`, `VisualID`) pair.
  - Removed "root" parameter from `CreateXlibSurfaceKHR()`, as it is redundant when a window on the same screen is specified as well.
  - Added appropriate X errors.
  - Adjusted wording of issue #1 and added agreed upon resolution.
- Revision 3, 2015-10-14 (Ian Elliott)
  - Renamed this extension from `VK_EXT_KHR_x11_surface` to `VK_EXT_KHR_xlib_surface`.
- Revision 4, 2015-10-26 (Ian Elliott)
  - Renamed from `VK_EXT_KHR_xlib_surface` to `VK_KHR_xlib_surface`.
- Revision 5, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to `vkCreateXlibSurfaceKHR`.
- Revision 6, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a `pCreateInfo` structure.

## `VK_EXT_acquire_xlib_display`

### Name String

`VK_EXT_acquire_xlib_display`

### Extension Type

Instance extension

### Registered Extension Number

90

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

- Requires [VK\\_EXT\\_direct\\_mode\\_display](#)

## Contact

- James Jones [Qcubanismo](#)

## Last Modified Date

2016-12-13

## IP Status

No known IP claims.

## Contributors

- Dave Airlie, Red Hat
- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Liam Middlebrook, NVIDIA
- Daniel Vetter, Intel

This extension allows an application to take exclusive control on a display currently associated with an X11 screen. When control is acquired, the display will be deassociated from the X11 screen until control is released or the specified display connection is closed. Essentially, the X11 screen will behave as if the monitor has been unplugged until control is released.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

- [vkAcquireXlibDisplayEXT](#)
- [vkGetRandROutputDisplayEXT](#)

## Issues

1) Should [vkAcquireXlibDisplayEXT](#) take an RandR display ID, or a Vulkan display handle as input?

**RESOLVED:** A Vulkan display handle. Otherwise there would be no way to specify handles to displays that had been “blacklisted” or prevented from being included in the X11 display list by some native platform or vendor-specific mechanism.

2) How does an application figure out which RandR display corresponds to a Vulkan display?

**RESOLVED:** A new function, `vkGetRandROutputDisplayEXT`, is introduced for this purpose.

3) Should `vkGetRandROutputDisplayEXT` be part of this extension, or a general Vulkan / RandR or Vulkan / Xlib extension?

**RESOLVED:** To avoid yet another extension, include it in this extension.

## Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

## VK\_EXT\_astc\_decode\_mode

### Name String

`VK_EXT_astc_decode_mode`

### Extension Type

Device extension

### Registered Extension Number

68

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Jan-Harald Fredriksen [janharaldfredriksen-arm](#)

### Last Modified Date

2018-08-07

### Contributors

- Jan-Harald Fredriksen, Arm

The existing specification requires that low dynamic range (LDR) ASTC textures are decompressed to FP16 values per component. In many cases, decompressing LDR textures to a lower precision intermediate result gives acceptable image quality. Source material for LDR textures is typically authored as 8-bit UNORM values, so decoding to FP16 values adds little value. On the other hand,

reducing precision of the decoded result reduces the size of the decompressed data, potentially improving texture cache performance and saving power.

The goal of this extension is to enable this efficiency gain on existing ASTC texture data. This is achieved by giving the application the ability to select the intermediate decoding precision.

Three decoding options are provided:

- Decode to `VK_FORMAT_R16G16B16A16_SFLOAT` precision: This is the default, and matches the required behavior in the core API.
- Decode to `VK_FORMAT_R8G8B8A8_UNORM` precision: This is provided as an option in LDR mode.
- Decode to `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32` precision: This is provided as an option in both LDR and HDR mode. In this mode, negative values cannot be represented and are clamped to zero. The alpha component is ignored, and the results are as if alpha was 1.0. This decode mode is optional and support can be queried via the physical device properties.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT`

## New Enums

None.

## New Structures

- `VkImageViewASTCDecodeModeEXT`
- `VkPhysicalDeviceASTCDecodeFeaturesEXT`

## New Functions

None.

## Issues

1) Are implementations allowed to decode at a higher precision than what is requested?

RESOLUTION: No.

If we allow this, then this extension could be exposed on all implementations that support ASTC.

But developers would have no way of knowing what precision was actually used, and thus whether the image quality is sufficient at reduced precision.

2) Should the decode mode be image view state and/or sampler state?

**RESOLUTION:** Image view state only.  
Some implementations treat the different decode modes as different texture formats.

## Example

Create an image view that decodes to `VK_FORMAT_R8G8B8A8_UNORM` precision:

```
VkImageViewASTCDecodeModeEXT decodeMode =  
{  
    VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT, // sType  
    NULL, // pNext  
    VK_FORMAT_R8G8B8A8_UNORM // decode mode  
};  
  
VkImageViewCreateInfo createInfo =  
{  
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO, // sType  
    &decodeMode, // pNext  
    // flags, image, viewType set to application-desired values  
    VK_FORMAT_ASTC_8x8_UNORM_BLOCK, // format  
    // components, subresourceRange set to application-desired values  
};  
  
VkImageView imageView;  
VkResult result = vkCreateImageView(  
    device,  
    &createInfo,  
    NULL,  
    &imageView);
```

## Version History

- Revision 1, 2018-08-07 (Jan-Harald Fredriksen)
  - Initial revision

## **VK\_EXT\_blend\_operation\_advanced**

### Name String

`VK_EXT_blend_operation_advanced`

### Extension Type

Device extension

### Registered Extension Number

149

## Revision

2

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Jeff Bolz [@jeffbolznv](#)

## Last Modified Date

2017-06-12

## Contributors

- Jeff Bolz, NVIDIA

This extension adds a number of “advanced” blending operations that **can** be used to perform new color blending operations, many of which are more complex than the standard blend modes provided by unextended Vulkan. This extension requires different styles of usage, depending on the level of hardware support and the enabled features:

- If `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT::advancedBlendCoherentOperations` is `VK_FALSE`, the new blending operations are supported, but a memory dependency **must** separate each advanced blend operation on a given sample. `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT` is used to synchronize reads using advanced blend operations.
- If `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT::advancedBlendCoherentOperations` is `VK_TRUE`, advanced blend operations obey primitive order just like basic blend operations.

In unextended Vulkan, the set of blending operations is limited, and **can** be expressed very simply. The `VK_BLEND_OP_MIN` and `VK_BLEND_OP_MAX` blend operations simply compute component-wise minimums or maximums of source and destination color components. The `VK_BLEND_OP_ADD`, `VK_BLEND_OP_SUBTRACT`, and `VK_BLEND_OP_REVERSE_SUBTRACT` modes multiply the source and destination colors by source and destination factors and either add the two products together or subtract one from the other. This limited set of operations supports many common blending operations but precludes the use of more sophisticated transparency and blending operations commonly available in many dedicated imaging APIs.

This extension provides a number of new “advanced” blending operations. Unlike traditional blending operations using `VK_BLEND_OP_ADD`, these blending equations do not use source and destination factors specified by `VkBlendFactor`. Instead, each blend operation specifies a complete equation based on the source and destination colors. These new blend operations are used for both RGB and alpha components; they **must** not be used to perform separate RGB and alpha blending (via different values of color and alpha `VkBlendOp`).

These blending operations are performed using premultiplied colors, where RGB colors **can** be considered premultiplied or non-premultiplied by alpha, according to the `srcPremultiplied` and `dstPremultiplied` members of `VkPipelineColorBlendAdvancedStateCreateInfoEXT`. If a color is considered non-premultiplied, the (R,G,B) color components are multiplied by the alpha component

prior to blending. For non-premultiplied color components in the range [0,1], the corresponding premultiplied color component would have values in the range  $[0 \times A, 1 \times A]$ .

Many of these advanced blending equations are formulated where the result of blending source and destination colors with partial coverage have three separate contributions: from the portions covered by both the source and the destination, from the portion covered only by the source, and from the portion covered only by the destination. The blend parameter `VkPipelineColorBlendAdvancedStateCreateInfoEXT::blendOverlap` can be used to specify a correlation between source and destination pixel coverage. If set to `VK_BLEND_OVERLAP_CONJOINT_EXT`, the source and destination are considered to have maximal overlap, as would be the case if drawing two objects on top of each other. If set to `VK_BLEND_OVERLAP_DISJOINT_EXT`, the source and destination are considered to have minimal overlap, as would be the case when rendering a complex polygon tessellated into individual non-intersecting triangles. If set to `VK_BLEND_OVERLAP_UNCORRELATED_EXT`, the source and destination coverage are assumed to have no spatial correlation within the pixel.

In addition to the coherency issues on implementations not supporting `advancedBlendCoherentOperations`, this extension has several limitations worth noting. First, the new blend operations have a limit on the number of color attachments they can be used with, as indicated by `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendMaxColorAttachments`. Additionally, blending precision may be limited to 16-bit floating-point, which may result in a loss of precision and dynamic range for framebuffer formats with 32-bit floating-point components, and in a loss of precision for formats with 12- and 16-bit signed or unsigned normalized integer components.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT`
- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`
- Extending `VkBlendOp`:
  - `VK_BLEND_OP_ZERO_EXT`
  - `VK_BLEND_OP_SRC_EXT`
  - `VK_BLEND_OP_DST_EXT`
  - `VK_BLEND_OP_SRC_OVER_EXT`
  - `VK_BLEND_OP_DST_OVER_EXT`
  - `VK_BLEND_OP_SRC_IN_EXT`
  - `VK_BLEND_OP_DST_IN_EXT`
  - `VK_BLEND_OP_SRC_OUT_EXT`

- VK\_BLEND\_OP\_DST\_OUT\_EXT
- VK\_BLEND\_OP\_SRC\_ATOP\_EXT
- VK\_BLEND\_OP\_DST\_ATOP\_EXT
- VK\_BLEND\_OP\_XOR\_EXT
- VK\_BLEND\_OP\_MULTIPLY\_EXT
- VK\_BLEND\_OP\_SCREEN\_EXT
- VK\_BLEND\_OP\_OVERLAY\_EXT
- VK\_BLEND\_OP\_DARKEN\_EXT
- VK\_BLEND\_OP\_LIGHTEN\_EXT
- VK\_BLEND\_OP\_COLORDODGE\_EXT
- VK\_BLEND\_OP\_COLORBURN\_EXT
- VK\_BLEND\_OP\_HARDLIGHT\_EXT
- VK\_BLEND\_OP\_SOFTLIGHT\_EXT
- VK\_BLEND\_OP\_DIFFERENCE\_EXT
- VK\_BLEND\_OP\_EXCLUSION\_EXT
- VK\_BLEND\_OP\_INVERT\_EXT
- VK\_BLEND\_OP\_INVERT\_RGB\_EXT
- VK\_BLEND\_OP\_LINEARDODGE\_EXT
- VK\_BLEND\_OP\_LINEARBURN\_EXT
- VK\_BLEND\_OP\_VIVIDLIGHT\_EXT
- VK\_BLEND\_OP\_LINEARLIGHT\_EXT
- VK\_BLEND\_OP\_PINLIGHT\_EXT
- VK\_BLEND\_OP\_HARDMIX\_EXT
- VK\_BLEND\_OP\_HSL\_HUE\_EXT
- VK\_BLEND\_OP\_HSL\_SATURATION\_EXT
- VK\_BLEND\_OP\_HSL\_COLOR\_EXT
- VK\_BLEND\_OP\_HSL\_LUMINOSITY\_EXT
- VK\_BLEND\_OP\_PLUS\_EXT
- VK\_BLEND\_OP\_PLUS\_CLAMPED\_EXT
- VK\_BLEND\_OP\_PLUS\_CLAMPED\_ALPHA\_EXT
- VK\_BLEND\_OP\_PLUS\_DARKER\_EXT
- VK\_BLEND\_OP\_MINUS\_EXT
- VK\_BLEND\_OP\_MINUS\_CLAMPED\_EXT
- VK\_BLEND\_OP\_CONTRAST\_EXT
- VK\_BLEND\_OP\_INVERT\_OVG\_EXT
- VK\_BLEND\_OP\_RED\_EXT
- VK\_BLEND\_OP\_GREEN\_EXT
- VK\_BLEND\_OP\_BLUE\_EXT

## New Enums

- [VkBlendOverlapEXT](#)

## New Structures

- [VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT](#)
- [VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT](#)
- [VkPipelineColorBlendAdvancedStateCreateInfoEXT](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2017-06-12 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2017-06-12 (Jeff Bolz)
  - Internal revisions

## VK\_EXT\_calibrated\_timestamps

### Name String

`VK_EXT_calibrated_timestamps`

### Extension Type

Device extension

### Registered Extension Number

185

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Daniel Rakos [@drakos-amd](#)

### Last Modified Date

2018-10-04

### IP Status

No known IP claims.

## Contributors

- Matthaeus G. Chajdas, AMD
- Alan Harrison, AMD
- Derrick Owens, AMD
- Daniel Rakos, AMD
- Jason Ekstrand, Intel
- Keith Packard, Valve

This extension provides an interface to query calibrated timestamps obtained quasi simultaneously from two time domains.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT`

## New Enums

- [VkTimeDomainEXT](#)

## New Structures

- [VkCalibratedTimestampInfoEXT](#)

## New Functions

- [vkGetPhysicalDeviceCalibrateableTimeDomainsEXT](#)
- [vkGetCalibratedTimestampsEXT](#)

## Issues

1) Is the device timestamp value returned in the same time domain as the timestamp values written by [vkCmdWriteTimestamp](#)?

**RESOLVED:** Yes.

2) What time domain is the host timestamp returned in?

**RESOLVED:** A query is provided to determine the calibrateable time domains. The expected host time domain used on Windows is that of QueryPerformanceCounter, and on Linux that of CLOCK\_MONOTONIC.

3) Should we support other time domain combinations than just one host and the device time

domain?

**RESOLVED:** Supporting that would need the application to query the set of supported time domains, while supporting only one host and the device time domain would only need a query for the host time domain type. The proposed API chooses the general approach for the sake of extensibility.

4) Shouldn't we use CLOCK\_MONOTONIC\_RAW instead of CLOCK\_MONOTONIC?

**RESOLVED:** CLOCK\_MONOTONIC is usable in a wider set of situations, however, it is subject to NTP adjustments so some use cases may prefer CLOCK\_MONOTONIC\_RAW. Thus this extension allows both to be exposed.

5) How can the application extrapolate future device timestamp values from the calibrated timestamp value?

**RESOLVED:** `VkPhysicalDeviceLimits::timestampPeriod` makes it possible to calculate future device timestamps as follows:

```
futureTimestamp = calibratedTimestamp + deltaNanoseconds / timestampPeriod
```

6) Can the host and device timestamp values drift apart over longer periods of time?

**RESOLVED:** Yes, especially as some time domains by definition allow for that to happen (e.g. CLOCK\_MONOTONIC is subject to NTP adjustments). Thus it's recommended that applications re-calibrate from time to time.

7) Should we add a query for reporting the maximum deviation of the timestamp values returned by calibrated timestamp queries?

**RESOLVED:** A global query seems inappropriate and difficult to enforce. However, it's possible to return the maximum deviation any single calibrated timestamp query can have by sampling one of the time domains twice as follows:

```
timestampX = timestampX_before = SampleTimeDomain(X)
for each time domain Y != X
    timestampY = SampleTimeDomain(Y)
timestampX_after = SampleTimeDomain(X)
maxDeviation = timestampX_after - timestampX_before
```

8) Can the maximum deviation reported ever be zero?

**RESOLVED:** Unless the tick of each clock corresponding to the set of time domains coincides and all clocks can literally be sampled simultaneously, there isn't really a possibility for the maximum deviation to be zero, so by convention the maximum deviation is always at least the maximum of the length of the ticks of the set of time domains calibrated and thus can never be zero.

## Version History

- Revision 1, 2018-10-04 (Daniel Rakos)
  - Internal revisions.

## VK\_EXT\_conditional\_rendering

### Name String

`VK_EXT_conditional_rendering`

### Extension Type

Device extension

### Registered Extension Number

82

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Vikram Kushwaha [@vkushwaha](#)

### Last Modified Date

2018-05-21

### IP Status

No known IP claims.

### Contributors

- Vikram Kushwaha, NVIDIA
- Daniel Rakos, AMD
- Jesse Hall, Google
- Jeff Bolz, NVIDIA
- Piers Daniell, NVIDIA
- Stuart Smith, Imagination Technologies

This extension allows the execution of one or more rendering commands to be conditional on a value in buffer memory. This may help an application reduce the latency by conditionally discarding rendering commands without application intervention. The conditional rendering commands are limited to draws, compute dispatches and clearing attachments within a conditional rendering block.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_COMMAND\\_BUFFER\\_INHERITANCE\\_CONDITIONAL\\_RENDERING\\_INFO\\_EXT](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_CONDITIONAL\\_RENDERING\\_FEATURES\\_EXT](#)
  - [VK\\_STRUCTURE\\_TYPE\\_CONDITIONAL\\_RENDERING\\_BEGIN\\_INFO\\_EXT](#)
- Extending [VkAccessFlagBits](#):
  - [VK\\_ACCESS\\_CONDITIONAL\\_RENDERING\\_READ\\_BIT\\_EXT](#)
- Extending [VkBufferUsageFlagBits](#):
  - [VK\\_BUFFER\\_USAGE\\_CONDITIONAL\\_RENDERING\\_BIT\\_EXT](#)
- Extending [VkPipelineStageFlagBits](#):
  - [VK\\_PIPELINE\\_STAGE\\_CONDITIONAL\\_RENDERING\\_BIT\\_EXT](#)

## New Enums

- [VkConditionalRenderingFlagBitsEXT](#)

## New Structures

- [VkConditionalRenderingBeginInfoEXT](#)
- Extending [VkCommandBufferInheritanceInfo](#):
  - [VkCommandBufferInheritanceConditionalRenderingInfoEXT](#)
- Extending [VkPhysicalDeviceFeatures2](#):
  - [VkPhysicalDeviceConditionalRenderingFeaturesEXT](#)

None.

## New Functions

- [vkCmdBeginConditionalRenderingEXT](#)
- [vkCmdEndConditionalRenderingEXT](#)

## Issues

1) Should conditional rendering affect copy and blit commands?

RESOLVED: Conditional rendering should not affect copies and blits.

2) Should secondary command buffers be allowed to execute while conditional rendering is active in the primary command buffer?

**RESOLVED:** The rendering commands in secondary command buffer will be affected by an active conditional rendering in primary command buffer if the `conditionalRenderingEnable` is set to `VK_TRUE`. Conditional rendering **must** not be active in the primary command buffer if `conditionalRenderingEnable` is `VK_FALSE`.

## Examples

None.

## Version History

- Revision 1, 2018-04-19 (Vikram Kushwaha)
  - First Version
- Revision 2, 2018-05-21 (Vikram Kushwaha)
  - Add new pipeline stage, access flags and limit conditional rendering to a subpass or entire renderpass.

## VK\_EXT\_conservative\_rasterization

### Name String

`VK_EXT_conservative_rasterization`

### Extension Type

Device extension

### Registered Extension Number

102

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Piers Daniell [Opdaniell-nv](#)

### Last Modified Data

2017-08-28

### Contributors

- Daniel Koch, NVIDIA
- Daniel Rakos, AMD
- Jeff Bolz, NVIDIA
- Slawomir Grajewski, Intel

- Stu Smith, Imagination Technologies

This extension adds a new rasterization mode called conservative rasterization. There are two modes of conservative rasterization; overestimation and underestimation.

When overestimation is enabled, if any part of the primitive, including its edges, covers any part of the rectangular pixel area, including its sides, then a fragment is generated with all coverage samples turned on. This extension allows for some variation in implementations by accounting for differences in overestimation, where the generating primitive size is increased at each of its edges by some sub-pixel amount to further increase conservative pixel coverage. Implementations can allow the application to specify an extra overestimation beyond the base overestimation the implementation already does. It also allows implementations to either cull degenerate primitives or rasterize them.

When underestimation is enabled, fragments are only generated if the rectangular pixel area is fully covered by the generating primitive. If supported by the implementation, when a pixel rectangle is fully covered the fragment shader input variable builtin called `FullyCoveredEXT` is set to true. The shader variable works in either overestimation or underestimation mode.

Implementations can process degenerate triangles and lines by either discarding them or generating conservative fragments for them. Degenerate triangles are those that end up with zero area after the rasterizer quantizes them to the fixed-point pixel grid. Degenerate lines are those with zero length after quantization.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT`

## New Enums

- `VkPipelineRasterizationConservativeStateCreateFlagsEXT`
- `VkConservativeRasterizationModeEXT`

## New Structures

- `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`
- `VkPipelineRasterizationConservativeStateCreateInfoEXT`

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2017-08-28 (Piers Daniell)
  - Internal revisions

## VK\_EXT\_debug\_utils

### Name String

`VK_EXT_debug_utils`

### Extension Type

Instance extension

### Registered Extension Number

129

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Mark Young [@marky-lunarg](#)

### Last Modified Date

2017-09-14

### Revision

1

### IP Status

No known IP claims.

### Dependencies

- This extension is written against version 1.0 of the Vulkan API.
- Requires `VkObjectType`

### Contributors

- Mark Young, LunarG
- Baldur Karlsson
- Ian Elliott, Google
- Courtney Goeltzenleuchter, Google

- Karl Schultz, LunarG
- Mark Lobodzinski, LunarG
- Mike Schuchardt, LunarG
- Jaakko Konttinen, AMD
- Dan Ginsburg, Valve Software
- Rolando Olivares, Epic Games
- Dan Baker, Oxide Games
- Kyle Spagnoli, NVIDIA
- Jon Ashburn, LunarG

Due to the nature of the Vulkan interface, there is very little error information available to the developer and application. By using the `VK_EXT_debug_utils` extension, developers **can** obtain more information. When combined with validation layers, even more detailed feedback on the application's use of Vulkan will be provided.

This extension provides the following capabilities:

- The ability to create a debug messenger which will pass along debug messages to an application supplied callback.
- The ability to identify specific Vulkan objects using a name or tag to improve tracking.
- The ability to identify specific sections within a `VkQueue` or `VkCommandBuffer` using labels to aid organization and offline analysis in external tools.

The main difference between this extension and `VK_EXT_debug_report` and `VK_EXT_debug_marker` is that those extensions use `VkDebugReportObjectTypeEXT` to identify objects. This extension uses the core `VkObjectType` in place of `VkDebugReportObjectTypeEXT`. The primary reason for this move is that no future object type handle enumeration values will be added to `VkDebugReportObjectTypeEXT` since the creation of `VkObjectType`.

In addition, this extension combines the functionality of both `VK_EXT_debug_report` and `VK_EXT_debug_marker` by allowing object name and debug markers (now called labels) to be returned to the application's callback function. This should assist in clarifying the details of a debug message including: what objects are involved and potentially which location within a `VkQueue` or `VkCommandBuffer` the message occurred.

## New Object Types

- `VkDebugUtilsMessengerEXT`

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT`
  - `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT`
  - `VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT`

- `VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT`
- `VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT`
- Extending `VkResult`:
  - `VK_ERROR_VALIDATION_FAILED_EXT`

## New Enums

- [VkDebugUtilsMessageSeverityFlagBitsEXT](#)
- [VkDebugUtilsMessageTypeFlagBitsEXT](#)

## New Structures

- [VkDebugUtilsObjectNameInfoEXT](#)
- [VkDebugUtilsObjectTagInfoEXT](#)
- [VkDebugUtilsLabelEXT](#)
- [VkDebugUtilsMessengerCallbackDataEXT](#)
- [VkDebugUtilsMessengerCreateInfoEXT](#)

## New Functions

- [vkSetDebugUtilsObjectNameEXT](#)
- [vkSetDebugUtilsObjectTagEXT](#)
- [vkQueueBeginDebugUtilsLabelEXT](#)
- [vkQueueEndDebugUtilsLabelEXT](#)
- [vkQueueInsertDebugUtilsLabelEXT](#)
- [vkCmdBeginDebugUtilsLabelEXT](#)
- [vkCmdEndDebugUtilsLabelEXT](#)
- [vkCmdInsertDebugUtilsLabelEXT](#)
- [vkCreateDebugUtilsMessengerEXT](#)
- [vkDestroyDebugUtilsMessengerEXT](#)
- [vkSubmitDebugUtilsMessageEXT](#)

## New Function Pointers

- [PFN\\_vkDebugUtilsMessengerCallbackEXT](#)

## Examples

### Example 1

`VK_EXT_debug_utils` allows an application to register multiple callbacks with any Vulkan component wishing to report debug information. Some callbacks may log the information to a file, others may

cause a debug break point or other application defined behavior. An application **can** register callbacks even when no validation layers are enabled, but they will only be called for loader and, if implemented, driver events.

To capture events that occur while creating or destroying an instance an application **can** link a `VkDebugUtilsMessengerCreateInfoEXT` structure to the `pNext` element of the `VkInstanceCreateInfo` structure given to `vkCreateInstance`. This callback is only valid for the duration of the `vkCreateInstance` and the `vkDestroyInstance` call. Use `vkCreateDebugUtilsMessengerEXT` to create persistent callback objects.

Example uses: Create three callback objects. One will log errors and warnings to the debug console using Windows `OutputDebugString`. The second will cause the debugger to break at that callback when an error happens and the third will log warnings to stdout.

```
extern VkInstance instance;
VkResult res;
VkDebugUtilsMessengerEXT cb1, cb2, cb3;

// Must call extension functions through a function pointer:
PFN_vkCreateDebugUtilsMessengerEXT pfnCreateDebugUtilsMessengerEXT =
(PFN_vkCreateDebugUtilsMessengerEXT)vkGetDeviceProcAddr(device,
"vkCreateDebugUtilsMessengerEXT");
PFN_vkDestroyDebugUtilsMessengerEXT pfnDestroyDebugUtilsMessengerEXT =
(PFN_vkDestroyDebugUtilsMessengerEXT)vkGetDeviceProcAddr(device,
"vkDestroyDebugUtilsMessengerEXT");

VkDebugUtilsMessengerCreateInfoEXT callback1 = {
    VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT, // sType
    NULL, // pNext
    0, // flags
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT | // messageSeverity
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT, // messageType
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | // myOutputDebugString,
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT, // pfnUserCallback
    NULL // pUserData
};
res = pfnCreateDebugUtilsMessengerEXT(instance, &callback1, NULL, &cb1);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}

callback1.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
callback1.pfnCallback = myDebugBreak;
callback1.pUserData = NULL;
res = pfnCreateDebugUtilsMessengerEXT(instance, &callback1, NULL, &cb2);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}
```

```

    }

VkDebugUtilsMessengerCreateInfoEXT callback3 = {
    VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT, // sType
    NULL, // pNext
    0, // flags
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT, // messageSeverity
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | // messageType
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT,
    mystdOutLogger, // pfnUserCallback
    NULL // pUserData
};

res = pfnCreateDebugUtilsMessengerEXT(instance, &callback3, NULL, &cb3);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}

...

// Remove callbacks when cleaning up
pfnDestroyDebugUtilsMessengerEXT(instance, cb1, NULL);
pfnDestroyDebugUtilsMessengerEXT(instance, cb2, NULL);
pfnDestroyDebugUtilsMessengerEXT(instance, cb3, NULL);

```

## Example 2

Associate a name with an image, for easier debugging in external tools or with validation layers that can print a friendly name when referring to objects in error messages.

```

extern VkDevice device;
extern VkImage image;

// Must call extension functions through a function pointer:
PFN_vkSetDebugUtilsObjectNameEXT pfnSetDebugUtilsObjectNameEXT =
(PFN_vkSetDebugUtilsObjectNameEXT)vkGetDeviceProcAddr(device,
"vkSetDebugUtilsObjectNameEXT");

// Set a name on the image
const VkDebugUtilsObjectNameInfoEXT imageNameInfo =
{
    VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT, // sType
    NULL, // pNext
    VK_OBJECT_TYPE_IMAGE, // objectType
    (uint64_t)image, // object
    "Brick Diffuse Texture", // pObjectName
};

pfnSetDebugUtilsObjectNameEXT(device, &imageNameInfo);

// A subsequent error might print:
// Image 'Brick Diffuse Texture' (0xc0dec0dedeadbeef) is used in a
// command buffer with no memory bound to it.

```

### Example 3

Annotating regions of a workload with naming information so that offline analysis tools can display a more usable visualization of the commands submitted.

```

extern VkDevice device;
extern VkCommandBuffer commandBuffer;

// Must call extension functions through a function pointer:
PFN_vkQueueBeginDebugUtilsLabelEXT pfnQueueBeginDebugUtilsLabelEXT =
(PFN_vkQueueBeginDebugUtilsLabelEXT)vkGetDeviceProcAddr(device,
"vkQueueBeginDebugUtilsLabelEXT");
PFN_vkQueueEndDebugUtilsLabelEXT pfnQueueEndDebugUtilsLabelEXT =
(PFN_vkQueueEndDebugUtilsLabelEXT)vkGetDeviceProcAddr(device,
"vkQueueEndDebugUtilsLabelEXT");
PFN_vkCmdBeginDebugUtilsLabelEXT pfnCmdBeginDebugUtilsLabelEXT =
(PFN_vkCmdBeginDebugUtilsLabelEXT)vkGetDeviceProcAddr(device,
"vkCmdBeginDebugUtilsLabelEXT");
PFN_vkCmdEndDebugUtilsLabelEXT pfnCmdEndDebugUtilsLabelEXT =
(PFN_vkCmdEndDebugUtilsLabelEXT)vkGetDeviceProcAddr(device,
"vkCmdEndDebugUtilsLabelEXT");
PFN_vkCmdInsertDebugUtilsLabelEXT pfnCmdInsertDebugUtilsLabelEXT =
(PFN_vkCmdInsertDebugUtilsLabelEXT)vkGetDeviceProcAddr(device,
"vkCmdInsertDebugUtilsLabelEXT");

```

```

// Describe the area being rendered
const VkDebugUtilsLabelEXT houseLabel =
{
    VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
    NULL, // pNext
    "Brick House", // pLabelName
    { 1.0f, 0.0f, 0.0f, 1.0f }, // color
};

// Start an annotated group of calls under the 'Brick House' name
pfnCmdBeginDebugUtilsLabelEXT(commandBuffer, &houseLabel);
{
    // A mutable structure for each part being rendered
    VkDebugUtilsLabelEXT housePartLabel =
    {
        VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
        NULL, // pNext
        NULL, // pLabelName
        { 0.0f, 0.0f, 0.0f, 0.0f }, // color
    };

    // Set the name and insert the marker
    housePartLabel.pLabelName = "Walls";
    pfnCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

    // Insert the drawcall for the walls
    vkCmdDrawIndexed(commandBuffer, 1000, 1, 0, 0, 0);

    // Insert a recursive region for two sets of windows
    housePartLabel.pLabelName = "Windows";
    pfnCmdBeginDebugUtilsLabelEXT(commandBuffer, &housePartLabel);
    {
        vkCmdDrawIndexed(commandBuffer, 75, 6, 1000, 0, 0);
        vkCmdDrawIndexed(commandBuffer, 100, 2, 1450, 0, 0);
    }
    pfnCmdEndDebugUtilsLabelEXT(commandBuffer);

    housePartLabel.pLabelName = "Front Door";
    pfnCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

    vkCmdDrawIndexed(commandBuffer, 350, 1, 1650, 0, 0);

    housePartLabel.pLabelName = "Roof";
    pfnCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

    vkCmdDrawIndexed(commandBuffer, 500, 1, 2000, 0, 0);
}
// End the house annotation started above
pfnCmdEndDebugUtilsLabelEXT(commandBuffer);

// Do other work

```

```

vkEndCommandBuffer(commandBuffer);

// Describe the queue being used
const VkDebugUtilsLabelEXT queueLabel =
{
    VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
    NULL, // pNext
    "Main Render Work", // pLabelName
    { 0.0f, 1.0f, 0.0f, 1.0f }, // color
};

// Identify the queue label region
pfnQueueBeginDebugUtilsLabelEXT(queue, &queueLabel);

// Submit the work for the main render thread
const VkCommandBuffer cmd_bufs[] = {commandBuffer};
VkSubmitInfo submit_info = {.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO,
                           .pNext = NULL,
                           .waitSemaphoreCount = 0,
                           .pWaitSemaphores = NULL,
                           .pWaitDstStageMask = NULL,
                           .commandBufferCount = 1,
                           .pCommandBuffers = cmd_bufs,
                           .signalSemaphoreCount = 0,
                           .pSignalSemaphores = NULL};
vkQueueSubmit(queue, 1, &submit_info, fence);

// End the queue label region
pfnQueueEndDebugUtilsLabelEXT(queue);

```

## Issues

1) Should we just name this extension [VK\\_EXT\\_debug\\_report2](#)

**RESOLVED:** No. There is enough additional changes to the structures to break backwards compatibility. So, a new name was decided that would not indicate any interaction with the previous extension.

2) Will validation layers immediately support all the new features.

**RESOLVED:** Not immediately. As one can imagine, there is a lot of work involved with converting the validation layer logging over to the new functionality. Basic logging, as seen in the origin [VK\\_EXT\\_debug\\_report](#) extension will be made available immediately. However, adding the labels and object names will take time. Since the priority for Khronos at this time is to continue focusing on Valid Usage statements, it may take a while before the new functionality is fully exposed.

3) If the validation layers won't expose the new functionality immediately, then what's the point of this extension?

**RESOLVED:** We needed a replacement for `VK_EXT_debug_report` because the `VkDebugReportObjectTypeEXT` enumeration will no longer be updated and any new objects will need to be debugged using the new functionality provided by this extension.

4) Should this extension be split into two separate parts (1 extension that is an instance extension providing the callback functionality, and another device extension providing the general debug marker and annotation functionality)?

**RESOLVED:** No, the functionality for this extension is too closely related. If we did split up the extension, where would the structures and enums live, and how would you define that the device behavior in the instance extension is really only valid if the device extension is enabled, and the functionality is passed in. It's cleaner to just define this all as an instance extension, plus it allows the application to enable all debug functionality provided with one enable string during `vkCreateInstance`.

## Version History

- Revision 1, 2017-09-14 (Mark Young and all listed Contributors)
  - Initial draft, based on `VK_EXT_debug_report` and `VK_EXT_debug_marker` in addition to previous feedback supplied from various companies including Valve, Epic, and Oxide games.

## `VK_EXT_depth_clip_enable`

### Name String

`VK_EXT_depth_clip_enable`

### Extension Type

Device extension

### Registered Extension Number

103

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Piers Daniell [Opdaniell-nv](#)

### Last Modified Data

2018-12-20

### Contributors

- Daniel Rakos, AMD
- Henri Verbeet, CodeWeavers
- Jeff Bolz, NVIDIA

- Philip Rebole, DXVK
- Tobias Hector, AMD

This extension allows the depth clipping operation, that is normally implicitly controlled by [VkPipelineRasterizationStateCreateInfo::depthClampEnable](#), to instead be controlled explicitly by [VkPipelineRasterizationDepthClipStateCreateInfoEXT::depthClipEnable](#).

This is useful for translating DX content which assumes depth clamping is always enabled, but depth clip can be controlled by the DepthClipEnable rasterization state (D3D12\_RASTERIZER\_DESC).

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_DEPTH\\_CLIP\\_ENABLE\\_FEATURES\\_EXT](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_RASTERIZATION\\_DEPTH\\_CLIP\\_STATE\\_CREATE\\_INFO\\_EXT](#)

## New Enums

- [VkPipelineRasterizationDepthClipStateCreateFlagsEXT](#)

## New Structures

- [VkPhysicalDeviceDepthClipEnableFeaturesEXT](#)
- [VkPipelineRasterizationDepthClipStateCreateInfoEXT](#)

## New Functions

None

## Issues

None

## Version History

- Revision 1, 2018-12-20 (Piers Daniell)
  - Internal revisions

## VK\_EXT\_depth\_range\_unrestricted

### Name String

[VK\\_EXT\\_depth\\_range\\_unrestricted](#)

## **Extension Type**

Device extension

## **Registered Extension Number**

14

## **Revision**

1

## **Extension and Version Dependencies**

- Requires Vulkan 1.0

## **Contact**

- Piers Daniell [Opdaniell-nv](#)

## **Last Modified Date**

2017-06-22

## **Contributors**

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

This extension removes the `VkViewport` `minDepth` and `maxDepth` restrictions that the values must be between `0.0` and `1.0`, inclusive. It also removes the same restriction on `VkPipelineDepthStencilStateCreateInfo` `minDepthBounds` and `maxDepthBounds`. Finally it removes the restriction on the `depth` value in `VkClearDepthStencilValue`.

## **New Object Types**

None.

## **New Enum Constants**

None.

## **New Enums**

None.

## **New Structures**

None.

## **New Functions**

None.

## Issues

1) How do `VkViewport` `minDepth` and `maxDepth` values outside of the `0.0` to `1.0` range interact with [Primitive Clipping](#)?

**RESOLVED:** The behavior described in [Primitive Clipping](#) still applies. If depth clamping is disabled the depth values are still clipped to  $0 \leq z_c \leq w_c$  before the viewport transform. If depth clamping is enabled the above equation is ignored and the depth values are instead clamped to the `VkViewport` `minDepth` and `maxDepth` values, which in the case of this extension can be outside of the `0.0` to `1.0` range.

2) What happens if a resulting depth fragment is outside of the `0.0` to `1.0` range and the depth buffer is fixed-point rather than floating-point?

**RESOLVED:** The supported range of a fixed-point depth buffer is `0.0` to `1.0` and depth fragments are clamped to this range.

## Version History

- Revision 1, 2017-06-22 (Piers Daniell)
  - Internal revisions

## VK\_EXT\_descriptor\_indexing

### Name String

`VK_EXT_descriptor_indexing`

### Extension Type

Device extension

### Registered Extension Number

162

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_maintenance3`

### Contact

- Jeff Bolz [@jeffbolznv](#)

### Status

Complete

### Last Modified Data

## Contributors

- Jeff Bolz, NVIDIA
- Daniel Rakos, AMD
- Slawomir Grajewski, Intel
- Tobias Hector, Imagination Technologies

This extension adds several small features which together enable applications to create large descriptor sets containing substantially all of their resources, and selecting amongst those resources with dynamic (non-uniform) indexes in the shader. There are feature enables and SPIR-V capabilities for non-uniform descriptor indexing in the shader, and non-uniform indexing in the shader requires use of a new `NonUniformEXT` decoration defined in the `SPV_EXT_descriptor_indexing` SPIR-V extension. There are descriptor set layout binding creation flags enabling several features:

- Descriptors can be updated after they are bound to a command buffer, such that the execution of the command buffer reflects the most recent update to the descriptors.
- Descriptors that are not used by any pending command buffers can be updated, which enables writing new descriptors for frame N+1 while frame N is executing.
- Relax the requirement that all descriptors in a binding that is “statically used” must be valid, such that descriptors that are not accessed by a submission need not be valid and can be updated while that submission is executing.
- The final binding in a descriptor set layout can have a variable size (and unsized arrays of resources are allowed in the `GL_EXT_nonuniform_qualifier` and `SPV_EXT_descriptor_indexing` extensions).

Note that it is valid for multiple descriptor arrays in a shader to use the same set and binding number, as long as they are all compatible with the descriptor type in the pipeline layout. This means a single array binding in the descriptor set can serve multiple texture dimensionalities, or an array of buffer descriptors can be used with multiple different block layouts.

There are new descriptor set layout and descriptor pool creation flags that are required to opt in to the update-after-bind functionality, and there are separate `maxPerStage*` and `maxDescriptorSet*` limits that apply to these descriptor set layouts which **may** be much higher than the pre-existing limits. The old limits only count descriptors in non-updateAfterBind descriptor set layouts, and the new limits count descriptors in all descriptor set layouts in the pipeline layout.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES_EXT`

- VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_DESCRIPTOR\_INDEXING\_PROPERTIES\_EXT
- VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_SET\_VARIABLE\_DESCRIPTOR\_COUNT\_ALLOCATE\_INFO\_EXT
- VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_SET\_VARIABLE\_DESCRIPTOR\_COUNT\_LAYOUT\_SUPPORT\_EXT
- Extending [VkDescriptorPoolCreateFlagBits](#):
  - VK\_DESCRIPTOR\_POOL\_CREATE\_UPDATE\_AFTER\_BIND\_BIT\_EXT
- Extending [VkDescriptorSetLayoutCreateFlagBits](#):
  - VK\_DESCRIPTOR\_SET\_LAYOUT\_CREATE\_UPDATE\_AFTER\_BIND\_POOL\_BIT\_EXT
- Extending [VkResult](#):
  - VK\_ERROR\_FRAGMENTATION\_EXT

## New Enums

- [VkDescriptorBindingFlagBitsEXT](#)

## New Structures

- [VkDescriptorSetLayoutBindingFlagsCreateInfoEXT](#)
- [VkPhysicalDeviceDescriptorIndexingFeaturesEXT](#)
- [VkPhysicalDeviceDescriptorIndexingPropertiesEXT](#)
- [VkDescriptorSetVariableDescriptorCountAllocateInfoEXT](#)
- [VkDescriptorSetVariableDescriptorCountLayoutSupportEXT](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2017-07-26 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2017-10-02 (Jeff Bolz)
  - ???

## VK\_EXT\_direct\_mode\_display

### Name String

`VK_EXT_direct_mode_display`

### Extension Type

Instance extension

## Registered Extension Number

89

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_display](#)

## Contact

- James Jones [@cubanismo](#)

## Last Modified Date

2016-12-13

## IP Status

No known IP claims.

## Contributors

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Liam Middlebrook, NVIDIA

This extension, along with related platform extensions, allows applications to take exclusive control of displays associated with a native windowing system. This is especially useful for virtual reality applications that wish to hide HMDs (head mounted displays) from the native platform's display management system, desktop, and/or other applications.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

- [vkReleaseDisplayEXT](#)

## Issues

1) Should this extension and its related platform-specific extensions leverage `VK_KHR_display`, or provide separate equivalent interfaces.

**RESOLVED:** Use `VK_KHR_display` concepts and objects. `VK_KHR_display` can be used to enumerate all displays on the system, including those attached to/in use by a window system or native platform, but `VK_KHR_display_swapchain` will fail to create a swapchain on in-use displays. This extension and its platform-specific children will allow applications to grab in-use displays away from window systems and/or native platforms, allowing them to be used with `VK_KHR_display_swapchain`.

2) Are separate calls needed to acquire displays and enable direct mode?

**RESOLVED:** No, these operations happen in one combined command. Acquiring a display puts it into direct mode.

## Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

## `VK_EXT_discard_rectangles`

### Name String

`VK_EXT_discard_rectangles`

### Extension Type

Device extension

### Registered Extension Number

100

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Piers Daniell [Opdaniell-nv](#)

### Last Modified Date

2016-12-22

### Interactions and External Dependencies

- Interacts with `VK_KHR_device_group`
- Interacts with Vulkan 1.1

## Contributors

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

This extension provides additional orthogonally aligned “discard rectangles” specified in framebuffer-space coordinates that restrict rasterization of all points, lines and triangles.

From zero to an implementation-dependent limit (specified by `maxDiscardRectangles`) number of discard rectangles can be operational at once. When one or more discard rectangles are active, rasterized fragments can either survive if the fragment is within any of the operational discard rectangles (`VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT` mode) or be rejected if the fragment is within any of the operational discard rectangles (`VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT` mode).

These discard rectangles operate orthogonally to the existing scissor test functionality. The discard rectangles can be different for each physical device in a device group by specifying the device mask and setting discard rectangle dynamic state.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT`
- Extending `VkDynamicState`
  - `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT`

## New Enums

- `VkPipelineDiscardRectangleStateCreateInfoEXT`
- `VkDiscardRectangleModeEXT`

## New Structures

- `VkPhysicalDeviceDiscardRectanglePropertiesEXT`
- `VkPipelineDiscardRectangleStateCreateInfoEXT`

## New Functions

- `vkCmdSetDiscardRectangleEXT`

## Issues

None.

## Version History

- Revision 1, 2016-12-22 (Piers Daniell)
  - Internal revisions

## VK\_EXT\_display\_control

### Name String

`VK_EXT_display_control`

### Extension Type

Device extension

### Registered Extension Number

92

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_EXT_display_surface_counter`
- Requires `VK_KHR_swapchain`

### Contact

- James Jones [@cubanismo](#)

### Last Modified Date

2016-12-13

### IP Status

No known IP claims.

### Contributors

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Daniel Vetter, Intel

This extension defines a set of utility functions for use with the `VK_KHR_display` and `VK_KHR_swapchain` extensions.

## New Enum Constants

- Extending `VkStructureType`:

- VK\_STRUCTURE\_TYPE\_DISPLAY\_POWER\_INFO\_EXT
- VK\_STRUCTURE\_TYPE\_DEVICE\_EVENT\_INFO\_EXT
- VK\_STRUCTURE\_TYPE\_DISPLAY\_EVENT\_INFO\_EXT
- VK\_STRUCTURE\_TYPE\_SWAPCHAIN\_COUNTER\_CREATE\_INFO\_EXT

## New Enums

- [VkDisplayPowerStateEXT](#)
- [VkDeviceEventTypeEXT](#)
- [VkDisplayEventTypeEXT](#)

## New Structures

- [VkDisplayPowerInfoEXT](#)
- [VkDeviceEventInfoEXT](#)
- [VkDisplayEventInfoEXT](#)
- [VkSwapchainCounterCreateInfoEXT](#)

## New Functions

- [vkDisplayPowerControlEXT](#)
- [vkRegisterDeviceEventEXT](#)
- [vkRegisterDisplayEventEXT](#)
- [vkGetSwapchainCounterEXT](#)

## Issues

1) Should this extension add an explicit “WaitForVsync” API or a fence signaled at vsync that the application can wait on?

**RESOLVED:** A fence. A separate API could later be provided that allows exporting the fence to a native object that could be inserted into standard run loops on POSIX and Windows systems.

2) Should callbacks be added for a vsync event, or in general to monitor events in Vulkan?

**RESOLVED:** No, fences should be used. Some events are generated by interrupts which are managed in the kernel. In order to use a callback provided by the application, drivers would need to have the userspace driver spawn threads that would wait on the kernel event, and hence the callbacks could be difficult for the application to synchronize with its other work given they would arrive on a foreign thread.

3) Should vblank or scanline events be exposed?

**RESOLVED:** Vblank events. Scanline events could be added by a separate extension, but the latency of processing an interrupt and waking up a userspace event is high enough that the accuracy of a scanline event would be rather low. Further, per-scanline interrupts are not supported by all

hardware.

## Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

## VK\_EXT\_display\_surface\_counter

### Name String

`VK_EXT_display_surface_counter`

### Extension Type

Instance extension

### Registered Extension Number

91

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_display](#)

### Contact

- James Jones [@cubanismo](#)

### Last Modified Date

2016-12-13

### IP Status

No known IP claims.

### Contributors

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Daniel Vetter, Intel

This extension defines a vertical blanking period counter associated with display surfaces. It provides a mechanism to query support for such a counter from a [VkSurfaceKHR](#) object.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT`

## New Enums

- [VkSurfaceCounterFlagsEXT](#)
- [VkSurfaceCounterFlagBitsEXT](#)

## New Structures

- [VkSurfaceCapabilities2EXT](#)

## New Functions

- [vkGetPhysicalDeviceSurfaceCapabilities2EXT](#)

## Issues

None.

## Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

## **VK\_EXT\_external\_memory\_dma\_buf**

### Name String

`VK_EXT_external_memory_dma_buf`

### Extension Type

Device extension

### Registered Extension Number

126

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_memory\\_fd](#)

### Contact

- Chad Versace [@chadversary](#)

## Last Modified Date

2017-10-10

## IP Status

No known IP claims.

## Contributors

- Chad Versace, Google
- James Jones, NVIDIA
- Jason Ekstrand, Intel

A `dma_buf` is a type of file descriptor, defined by the Linux kernel, that allows sharing memory across kernel device drivers and across processes. This extension enables applications to import a `dma_buf` as `VkDeviceMemory`, to export `VkDeviceMemory` as a `dma_buf`, and to create `VkBuffer` objects that **can** be bound to that memory.

## New Enum Constants

- Extending `VkExternalMemoryHandleTypeFlagBitsKHR`:
  - `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT`

## Issues

1) How does the application, when creating a `VkImage` that it intends to bind to `dma_buf` `VkDeviceMemory` containing an externally produced image, specify the memory layout (such as row pitch and DRM format modifier) of the `VkImage`? In other words, how does the application achieve behavior comparable to that provided by `EGL_EXT_image_dma_buf_import` and `EGL_EXT_image_dma_buf_import_modifiers`?

**RESOLVED:** Features comparable to those in `EGL_EXT_image_dma_buf_import` and `EGL_EXT_image_dma_buf_import_modifiers` will be provided by an extension layered atop this one.

2) Without the ability to specify the memory layout of external `dma_buf` images, how is this extension useful?

**RESOLVED:** This extension provides exactly one new feature: the ability to import/export between `dma_buf` and `VkDeviceMemory`. This feature, together with features provided by `VK_KHR_external_memory_fd`, is sufficient to bind a `VkBuffer` to `dma_buf`.

## Version History

- Revision 1, 2017-10-10 (Chad Versace)
  - Squashed internal revisions

## VK\_EXT\_external\_memory\_host

### Name String

`VK_EXT_external_memory_host`

## Extension Type

Device extension

## Registered Extension Number

179

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_memory](#)

## Contact

- Daniel Rakos [@drakos-amd](#)

## Last Modified Date

2017-11-10

## IP Status

No known IP claims.

## Contributors

- Jaakko Konttinen, AMD
- David Mao, AMD
- Daniel Rakos, AMD
- Tobias Hector, Imagination Technologies
- Jason Ekstrand, Intel
- James Jones, NVIDIA

This extension enables an application to import host allocations and host mapped foreign device memory to Vulkan memory objects.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT`
  - `VK_STRUCTURE_TYPE_MEMORY_HOST_POINTER_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT`
- Extending [VkExternalMemoryHandleTypeFlagBitsKHR](#):
  - `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT`

◦ `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`

## New Enums

None.

## New Structs

- `VkImportMemoryHostPointerInfoEXT`
- `VkMemoryHostPointerPropertiesEXT`
- `VkPhysicalDeviceExternalMemoryHostPropertiesEXT`

## New Functions

- `vkGetMemoryHostPointerPropertiesEXT`

## Issues

1) What memory type has to be used to import host pointers?

RESOLVED: Depends on the implementation. Applications have to use the new `vkGetMemoryHostPointerPropertiesEXT` command to query the supported memory types for a particular host pointer. The reported memory types may include memory types that come from a memory heap that is otherwise not usable for regular memory object allocation and thus such a heap's size may be zero.

2) Can the application still access the contents of the host allocation after importing?

RESOLVED: Yes. However, usual synchronization requirements apply.

3) Can the application free the host allocation?

RESOLVED: No, it violates valid usage conditions. Using the memory object imported from a host allocation that's already freed thus results in undefined behavior.

4) Is `vkMapMemory` expected to return the same host address which was specified when importing it to the memory object?

RESOLVED: No. Implementations are allowed to return the same address but it's not required. Some implementations might return a different virtual mapping of the allocation, although the same physical pages will be used.

5) Is there any limitation on the alignment of the host pointer and/or size?

RESOLVED: Yes. Both the address and the size have to be an integer multiple of `minImportedHostPointerAlignment`. In addition, some platforms and foreign devices may have additional restrictions.

6) Can the same host allocation be imported multiple times into a given physical device?

RESOLVED: No, at least not guaranteed by this extension. Some platforms do not allow locking the same physical pages for device access multiple times, so attempting to do it may result in undefined behavior.

7) Does this extension support exporting the new handle type?

RESOLVED: No.

8) Should we include the possibility to import host mapped foreign device memory using this API?

RESOLVED: Yes, through a separate handle type. Implementations are still allowed to support only one of the handle types introduced by this extension by not returning import support for a particular handle type as returned in [VkExternalMemoryPropertiesKHR](#).

## Version History

- Revision 1, 2017-11-10 (Daniel Rakos)
  - Internal revisions

## VK\_EXT\_filter\_cubic

### Name String

`VK_EXT_filter_cubic`

### Extension Type

Device extension

### Registered Extension Number

171

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_IMG\\_filter\\_cubic](#)

### Contact

- Bill Licea-Kane [@wwlk](#)

### Last Modified Date

2019-01-24

### Contributors

- Bill Licea-Kane, Qualcomm Technologies, Inc.
- Andrew Garrard, Samsung
- Daniel Koch, NVIDIA

- Donald Scorgie, Imagination Technologies
- Graeme Leese, Broadcom
- Jan-Herald Fredericksen, ARM
- Jeff Leger, Qualcomm Technologies, Inc.
- Tobias Hector, AMD
- Tom Olson, ARM
- Stuart Smith, Imagination Technologies

`VK_EXT_filter_cubic` extends `VK_IMG_filter_cubic`.

It documents cubic filtering of other image view types. It adds new structures that **can** be added to the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` and `VkImageFormatProperties2` that **can** be used to determine which image types and which image view types support cubic filtering.

## New Structures

- `VkPhysicalDeviceImageViewImageFormatInfoEXT`
- `VkFilterCubicImageViewImageFormatPropertiesEXT`

## Version History

- Revision 2, 2019-06-05 (wwlk)
  - Clarify 1D optional
- Revision 1, 2019-01-24 (wwlk)
  - Initial version

## VK\_EXT\_fragment\_density\_map

### Name String

`VK_EXT_fragment_density_map`

### Extension Type

Device extension

### Registered Extension Number

219

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

## Contact

- Matthew Netsch [@mnetsch](#)

## Last Modified Date

2018-09-25

## Interactions and External Dependencies

- This extension requires the `SPV_EXT_fragment_invocation_density` SPIR-V extension.

## Contributors

- Matthew Netsch, Qualcomm Technologies, Inc.
- Robert VanReenen, Qualcomm Technologies, Inc.
- Jonathan Wicks, Qualcomm Technologies, Inc.
- Tate Hornbeck, Qualcomm Technologies, Inc.
- Sam Holmes, Qualcomm Technologies, Inc.
- Jeff Leger, Qualcomm Technologies, Inc.
- Jan-Harald Fredriksen, ARM
- Jeff Bolz, NVIDIA
- Pat Brown, NVIDIA
- Daniel Rakos, AMD
- Piers Daniell, NVIDIA

This extension allows an application to specify areas of the render target where the fragment shader may be invoked fewer times. These fragments are broadcasted out to multiple pixels to cover the render target.

The primary use of this extension is to reduce workloads in areas where lower quality may not be perceived such as the distorted edges of a lens or the periphery of a user's gaze.

## New Object Types

None.

## New Enum Constants

- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT`
- Extending `VkFormatFeatureFlagBits`:
  - `VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT`
- Extending `VkImageCreateFlagBits`:
  - `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`
- Extending `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT`

- Extending [VkImageUsageFlagBits](#):
  - `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`
- Extending [VkImageViewCreateFlagBits](#):
  - `VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT`
- Extending [VkPipelineStageFlagBits](#):
  - `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`
- Extending [VkSamplerCreateFlagBits](#):
  - `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`
  - `VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT`
- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_RENDER_PASS_FRAGMENT_DENSITY_MAP_CREATE_INFO_EXT`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceFragmentDensityMapFeaturesEXT](#)
- [VkPhysicalDeviceFragmentDensityMapPropertiesEXT](#)
- [VkRenderPassFragmentDensityMapCreateInfoEXT](#)

## New Functions

None.

## New or Modified Built-In Variables

- `FragInvocationCountEXT`
- `FragSizeEXT`

## New Variable Decorations

None.

## New SPIR-V Capabilities

- [FragmentDensityEXT](#)

## Version History

- Revision 1, 2018-09-25 (Matthew Netsch)
  - Initial version

# **VK\_EXT\_fragment\_shader\_interlock**

## **Name String**

`VK_EXT_fragment_shader_interlock`

## **Extension Type**

Device extension

## **Registered Extension Number**

252

## **Revision**

1

## **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

## **Contact**

- Piers Daniell [Opdaniell-nv](#)

## **Last Modified Data**

2019-05-02

## **Interactions and External Dependencies**

- This extension requires the `SPV_EXT_fragment_shader_interlock` SPIR-V extension.
- This extension requires the `GL_ARB_fragment_shader_interlock`, extensions for GLSL source languages.

## **Contributors**

- Daniel Koch, NVIDIA
- Graeme Leese, Broadcom
- Jan-Harald Fredriksen, Arm
- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA
- Ruihao Zhang, Qualcomm
- Slawomir Grajewski, Intel
- Spencer Fricke, Samsung

This extension adds support for the `FragmentShaderPixelInterlockEXT`, `FragmentShaderSampleInterlockEXT`, and `FragmentShaderShadingRateInterlockEXT` capabilities from the `SPV_EXT_fragment_shader_interlock` extension to Vulkan.

Enabling these capabilities provides a critical section for fragment shaders to avoid overlapping pixels being processed at the same time, and certain guarantees about the ordering of fragment

shader invocations of fragments of overlapping pixels.

This extension can be useful for algorithms that need to access per-pixel data structures via shader loads and stores. Algorithms using this extension can access per-pixel data structures in critical sections without other invocations accessing the same per-pixel data. Additionally, the ordering guarantees are useful for cases where the API ordering of fragments is meaningful. For example, applications may be able to execute programmable blending operations in the fragment shader, where the destination buffer is read via image loads and the final value is written via image stores.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT`

## New Enums

None.

## New Structures

- Extending [VkPhysicalDeviceFeatures2](#):
  - [VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT](#)

## New Functions

None.

## New SPIR-V Capabilities

- [FragmentShaderInterlockEXT](#)
- [FragmentShaderPixelInterlockEXT](#)
- [FragmentShaderShadingRateInterlockEXT](#)

## Issues

None.

## Version History

- Revision 1, 2019-05-24 (Piers Daniell)
  - Internal revisions

## **VK\_EXT\_full\_screen\_exclusive**

### **Name String**

`VK_EXT_full_screen_exclusive`

### **Extension Type**

Device extension

### **Registered Extension Number**

256

### **Revision**

4

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_surface`
- Requires `VK_KHR_get_surface_capabilities2`
- Requires `VK_KHR_swapchain`

### **Contact**

- James Jones [@cubanismo](#)

### **Last Modified Date**

2019-03-12

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

- Interacts with `VK_KHR_device_group`, `VK_KHR_win32_surface`, and Vulkan 1.1

### **Contributors**

- Hans-Kristian Arntzen, ARM
- Slawomir Grajewski, Intel
- Tobias Hector, AMD
- James Jones, NVIDIA
- Daniel Rakos, AMD
- Jeff Juliano, NVIDIA
- Joshua Schnarr, NVIDIA
- Aaron Hagan, AMD

This extension allows applications to set the policy for swapchain creation and presentation

mechanisms relating to full-screen access. Implementations may be able to acquire exclusive access to a particular display for an application window that covers the whole screen. This can increase performance on some systems by bypassing composition, however it can also result in disruptive or expensive transitions in the underlying windowing system when a change occurs.

Applications can choose between explicitly disallowing or allowing this behavior, letting the implementation decide, or managing this mode of operation directly using the new `vkAcquireFullScreenExclusiveModeEXT` and `vkReleaseFullScreenExclusiveModeEXT` commands.

## New Enum Constants

- Extending `VkResult`
  - `VK_ERROR_FULLSCREEN_EXCLUSIVE_MODE_LOST_EXT`
- Extending `VkStructureType`
  - `VK_STRUCTURE_TYPE_SURFACE_FULLSCREEN_EXCLUSIVE_INFO_EXT`
  - `VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_FULLSCREEN_EXCLUSIVE_EXT`
  - `VK_STRUCTURE_TYPE_SURFACE_FULLSCREEN_EXCLUSIVE_WIN32_INFO_EXT`

## New Enums

- `VkFullScreenExclusiveEXT`

## New Structures

- `VkSurfaceFullScreenExclusiveInfoEXT`
- `VkSurfaceCapabilitiesFullScreenExclusiveEXT`
- `VkSurfaceFullScreenExclusiveWin32InfoEXT`

## New Functions

- `vkGetPhysicalDeviceSurfacePresentModes2EXT`
- `vkGetDeviceGroupSurfacePresentModes2EXT`
- `vkAcquireFullScreenExclusiveModeEXT`
- `vkReleaseFullScreenExclusiveModeEXT`

## Issues

1) What should the extension & flag be called?

RESOLVED: `VK_EXT_full_screen_exclusive`.

Other options considered (prior to the app-controlled mode) were:

- `VK_EXT_smooth_fullscreen_transition`
- `VK_EXT_fullscreen_behavior`
- `VK_EXT_fullscreen_preference`

- VK\_EXT\_fullscreen\_hint
- VK\_EXT\_fastFullscreen\_transition
- VK\_EXT\_avoidFullscreen\_exclusive

2) Do we need more than a boolean toggle?

RESOLVED: Yes.

Using an enum with default/allowed/disallowed/app-controlled enables applications to accept driver default behavior, specifically override it in either direction without implying the driver is ever required to use full-screen exclusive mechanisms, or manage this mode explicitly.

3) Should this be a KHR or EXT extension?

RESOLVED: EXT, in order to allow it to be shipped faster.

4) Can the fullscreen hint affect the surface capabilities, and if so, should the hint also be specified as input when querying the surface capabilities?

RESOLVED: Yes on both accounts.

While the hint does not guarantee a particular fullscreen mode will be used when the swapchain is created, it can sometimes imply particular modes will NOT be used. If the driver determines that it will opt-out of using a particular mode based on the policy, and knows it can only support certain capabilities if that mode is used, it would be confusing at best to the application to report those capabilities in such cases. Not allowing implementations to report this state to applications could result in situations where applications are unable to determine why swapchain creation fails when they specify certain hint values, which could result in never-terminating surface creation loops.

5) Should full-screen be one word or two?

RESOLVED: Two words.

"Fullscreen" is not in my dictionary, and web searches did not turn up definitive proof that it is a colloquially accepted compound word. Documentation for the corresponding Windows API mechanisms dithers. The text consistently uses a hyphen, but none-the-less, there is a SetFullscreenState method in the DXGI swapchain object. Given this inconclusive external guidance, it is best to adhere to the Vulkan style guidelines and avoid inventing new compound words.

## Version History

- Revision 4, 2019-03-12 (Tobias Hector)
  - Added application-controlled mode, and related functions
  - Tidied up appendix
- Revision 3, 2019-01-03 (James Jones)
  - Renamed to VK\_EXT\_full\_screen\_exclusive
  - Made related adjustments to the tri-state enumerant names.

- Revision 2, 2018-11-27 (James Jones)
  - Renamed to VK\_KHR\_fullscreen\_behavior
  - Switched from boolean flag to tri-state enum
- Revision 1, 2018-11-06 (James Jones)
  - Internal revision

## **VK\_EXT\_global\_priority**

### **Name String**

`VK_EXT_global_priority`

### **Extension Type**

Device extension

### **Registered Extension Number**

175

### **Revision**

2

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Contact**

- Andres Rodriguez [@lostgoat](#)

### **Last Modified Date**

2017-10-06

### **IP Status**

No known IP claims.

### **Contributors**

- Andres Rodriguez, Valve
- Pierre-Loup Griffais, Valve
- Dan Ginsburg, Valve
- Mitch Singer, AMD

In Vulkan, users can specify device-scope queue priorities. In some cases it may be useful to extend this concept to a system-wide scope. This extension provides a mechanism for caller's to set their system-wide priority. The default queue priority is `VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT`.

The driver implementation will attempt to skew hardware resource allocation in favour of the higher-priority task. Therefore, higher-priority work may retain similar latency and throughput characteristics even if the system is congested with lower priority work.

The global priority level of a queue shall take precedence over the per-process queue priority ([VkDeviceQueueCreateInfo::pQueuePriorities](#)).

Abuse of this feature may result in starving the rest of the system from hardware resources. Therefore, the driver implementation may deny requests to acquire a priority above the default priority ([VK\\_QUEUE\\_GLOBAL\\_PRIORITY\\_MEDIUM\\_EXT](#)) if the caller does not have sufficient privileges. In this scenario [VK\\_ERROR\\_NOT\\_PERMITTED\\_EXT](#) is returned.

The driver implementation may fail the queue allocation request if resources required to complete the operation have been exhausted (either by the same process or a different process). In this scenario [VK\\_ERROR\\_INITIALIZATION\\_FAILED](#) is returned.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_DEVICE\\_QUEUE\\_GLOBAL\\_PRIORITY\\_CREATE\\_INFO\\_EXT](#)
- Extending [VkResult](#):
  - [VK\\_ERROR\\_NOT\\_PERMITTED\\_EXT](#)

## New Enums

- [VkQueueGlobalPriorityEXT](#)

## New Structures

- [VkDeviceQueueGlobalPriorityCreateInfoEXT](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 2, 2017-11-03 (Andres Rodriguez)
  - Fixed [VkQueueGlobalPriorityEXT](#) missing \_EXT suffix
- Revision 1, 2017-10-06 (Andres Rodriguez)
  - First version.

## **VK\_EXT\_hdr\_metadata**

### **Name String**

`VK_EXT_hdr_metadata`

### **Extension Type**

Device extension

### **Registered Extension Number**

106

### **Revision**

2

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_swapchain](#)

### **Contact**

- Courtney Goeltzenleuchter [Courtney-g](#)

### **Last Modified Date**

2018-12-19

### **IP Status**

No known IP claims.

### **Contributors**

- Courtney Goeltzenleuchter, Google

This extension defines two new structures and a function to assign SMPTE (the Society of Motion Picture and Television Engineers) 2086 metadata and CTA (Consumer Technology Association) 861.3 metadata to a swapchain. The metadata includes the color primaries, white point, and luminance range of the mastering display, which all together define the color volume that contains all the possible colors the mastering display can produce. The mastering display is the display where creative work is done and creative intent is established. To preserve such creative intent as much as possible and achieve consistent color reproduction on different viewing displays, it is useful for the display pipeline to know the color volume of the original mastering display where content was created or tuned. This avoids performing unnecessary mapping of colors that are not displayable on the original mastering display. The metadata also includes the `maxContentLightLevel` and `maxFrameAverageLightLevel` as defined by CTA 861.3.

While the general purpose of the metadata is to assist in the transformation between different color volumes of different displays and help achieve better color reproduction, it is not in the scope of this extension to define how exactly the metadata should be used in such a process. It is up to the implementation to determine how to make use of the metadata.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_HDR_METADATA_EXT`

## New Structures

- `VkXYColorEXT`
- `VkHdrMetadataEXT`

## New Functions

- `vkSetHdrMetadataEXT`

## Issues

1) Do we need a query function?

**PROPOSED:** No, Vulkan does not provide queries for state that the application can track on its own.

2) Should we specify default if not specified by the application?

**PROPOSED:** No, that leaves the default up to the display.

## Version History

- Revision 1, 2016-12-27 (Courtney Goeltzenleuchter)
  - Initial version
- Revision 2, 2018-12-19 (Courtney Goeltzenleuchter)
  - Correct implicit validity for `VkHdrMetadataEXT` structure

## `VK_EXT_headless_surface`

### Name String

`VK_EXT_headless_surface`

### Extension Type

Instance extension

### Registered Extension Number

257

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

- Requires [VK\\_KHR\\_surface](#)

## Contact

- Lisa Wu [Qchengtianww](#)

## Last Modified Date

2019-03-21

## IP Status

No known IP claims.

## Contributors

- Ray Smith, Arm

The [VK\\_EXT\\_headless\\_surface](#) extension is an instance extension. It provides a mechanism to create [VkSurfaceKHR](#) objects independently of any window system or display device. The presentation operation for a swapchain created from a headless surface is by default a no-op, resulting in no externally-visible result.

Because there is no real presentation target, future extensions can layer on top of the headless surface to introduce arbitrary or customisable sets of restrictions or features. These could include features like saving to a file or restrictions to emulate a particular presentation target.

This functionality is expected to be useful for application and driver development because it allows any platform to expose an arbitrary or customisable set of restrictions and features of a presentation engine. This makes it a useful portable test target for applications targeting a wide range of presentation engines where the actual target presentation engines might be scarce, unavailable or otherwise undesirable or inconvenient to use for general Vulkan application development.

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_HEADLESS\\_SURFACE\\_CREATE\\_INFO\\_EXT](#)

## New Enums

None

## New Structures

- [VkHeadlessSurfaceCreateInfoEXT](#)

## New Functions

- [vkCreateHeadlessSurfaceEXT](#)

## Issues

None

## Version History

- Revision 1, 2019-03-21 (Ray Smith)
  - Initial draft

## VK\_EXT\_host\_query\_reset

### Name String

[VK\\_EXT\\_host\\_query\\_reset](#)

### Extension Type

Device extension

### Registered Extension Number

262

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Bas Nieuwenhuizen [OBNieuwenhuizen](#)

### Last Modified Date

2019-03-06

### IP Status

No known IP claims.

### Interactions and External Dependencies

### Contributors

- Bas Nieuwenhuizen, Google
- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA
- Piers Daniell, NVIDIA

This extension adds a new function to reset queries from the host.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES_EXT`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceHostQueryResetFeaturesEXT](#)

## New Functions

- [vkResetQueryPoolEXT](#)

## Issues

## Version History

- Revision 1, 2019-03-12 (Bas Nieuwenhuizen)
  - Initial draft

## VK\_EXT\_image\_drm\_format\_modifier

### Name String

`VK_EXT_image_drm_format_modifier`

### Extension Type

Device extension

### Registered Extension Number

159

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_bind\\_memory2](#)
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)
- Requires [VK\\_KHR\\_image\\_format\\_list](#)
- Requires [VK\\_KHR\\_sampler\\_ycbcr\\_conversion](#)

## Contact

- Chad Versace [@chadversary](#)

## Last Modified Date

2018-08-29

## IP Status

No known IP claims.

## Contributors

- Antoine Labour, Google
- Bas Nieuwenhuizen, Google
- Chad Versace, Google
- James Jones, NVIDIA
- Jason Ekstrand, Intel
- Jörg Wagner, ARM
- Kristian Høgsberg Kristensen, Google
- Ray Smith, ARM

## Overview

### Summary

This extension provides the ability to use *DRM format modifiers* with images, enabling Vulkan to better integrate with the Linux ecosystem of graphics, video, and display APIs.

Its functionality closely overlaps with [EGL\\_EXT\\_image\\_dma\\_buf\\_import\\_modifiers<sup>2</sup>](#) and [EGL\\_MESA\\_image\\_dma\\_buf\\_export<sup>3</sup>](#). Unlike the EGL extensions, this extension does not require the use of a specific handle type (such as a `dma_buf`) for external memory and provides more explicit control of image creation.

### Introduction to DRM Format Modifiers

A *DRM format modifier* is a 64-bit, vendor-prefixed, semi-opaque unsigned integer. Most *modifiers* represent a concrete, vendor-specific tiling format for images. Some exceptions are `DRM_FORMAT_MOD_LINEAR` (which is not vendor-specific); `DRM_FORMAT_MOD_NONE` (which is an alias of `DRM_FORMAT_MOD_LINEAR` due to historical accident); and `DRM_FORMAT_MOD_INVALID` (which does not represent a tiling format). The *modifier*'s vendor prefix consists of the 8 most significant bits. The canonical list of *modifiers* and vendor prefixes is found in `drm_fourcc.h` in the Linux kernel source. The other dominant source of *modifiers* are vendor kernel trees.

One goal of *modifiers* in the Linux ecosystem is to enumerate for each vendor a reasonably sized set of tiling formats that are appropriate for images shared across processes, APIs, and/or devices, where each participating component may possibly be from different vendors. A non-goal is to enumerate all tiling formats supported by all vendors. Some tiling formats used internally by vendors are inappropriate for sharing; no *modifiers* should be assigned to such tiling formats.

Modifier values typically do not *describe* memory layouts. More precisely, a *modifier*'s lower 56 bits usually have no structure. Instead, modifiers *name* memory layouts; they name a small set of vendor-preferred layouts for image sharing. As a consequence, in each vendor namespace the modifier values are often sequentially allocated starting at 1.

Each *modifier* is usually supported by a single vendor and its name matches the pattern `{VENDOR}_FORMAT_MOD_*` or `DRM_FORMAT_MOD_{VENDOR}_*`. Examples are `I915_FORMAT_MOD_X_TILED` and `DRM_FORMAT_MOD_BROADCOM_VC4_T_TILED`. An exception is `DRM_FORMAT_MOD_LINEAR`, which is supported by most vendors.

Many APIs in Linux use *modifiers* to negotiate and specify the memory layout of shared images. For example, a Wayland compositor and Wayland client may, by relaying *modifiers* over the Wayland protocol `zwp_linux_dmabuf_v1`, negotiate a vendor-specific tiling format for a shared `wl_buffer`. The client may allocate the underlying memory for the `wl_buffer` with GBM, providing the chosen *modifier* to `gbm_bo_create_with_modifiers`. The client may then import the `wl_buffer` into Vulkan for producing image content, providing the resource's `dma_buf` to `VkImportMemoryFdInfoKHR` and its *modifier* to `VkImageDrmFormatModifierExplicitCreateInfoEXT`. The compositor may then import the `wl_buffer` into OpenGL for sampling, providing the resource's `dma_buf` and *modifier* to `eglCreateImage`. The compositor may also bypass OpenGL and submit the `wl_buffer` directly to the kernel's display API, providing the `dma_buf` and *modifier* through `drm_mode_fb_cmd2`.

## Format Translation

*Modifier*-capable APIs often pair *modifiers* with DRM formats, which are defined in `drm_fourcc.h`. However, `VK_EXT_image_drm_format_modifier` uses `VkFormat` instead of DRM formats. The application must convert between `VkFormat` and DRM format when it sends or receives a DRM format to or from an external API.

The mapping from `VkFormat` to DRM format is lossy. Therefore, when receiving a DRM format from an external API, often the application must use information from the external API to accurately map the DRM format to a `VkFormat`. For example, DRM formats do not distinguish between RGB and sRGB (as of 2018-03-28); external information is required to identify the image's colorspace.

The mapping between `VkFormat` and DRM format is also incomplete. For some DRM formats there exist no corresponding Vulkan format, and for some Vulkan formats there exist no corresponding DRM format.

## Usage Patterns

Three primary usage patterns are intended for this extension:

- **Negotiation.** The application negotiates with *modifier*-aware, external components to determine sets of image creation parameters supported among all components.

In the Linux ecosystem, the negotiation usually assumes the image is a 2D, single-sampled, non-mipmapped, non-array image; this extension permits that assumption but does not require it. The result of the negotiation usually resembles a set of tuples such as `(drmFormat, drmFormatModifier)`, where each participating component supports all tuples in the set.

Many details of this negotiation—such as the protocol used during negotiation, the set of image

creation parameters expressable in the protocol, and how the protocol chooses which process and which API will create the image—are outside the scope of this specification.

In this extension, `vkGetPhysicalDeviceFormatProperties2` with `VkDrmFormatModifierPropertiesListEXT` serves a primary role during the negotiation, and `vkGetPhysicalDeviceImageFormatProperties2` with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` serves a secondary role.

- **Import.** The application imports an image with a *modifier*.

In this pattern, the application receives from an external source the image's memory and its creation parameters, which are often the result of the negotiation described above. Some image creation parameters are implicitly defined by the external source; for example, `VK_IMAGE_TYPE_2D` is often assumed. Some image creation parameters are usually explicit, such as the image's `format`, `drmFormatModifier`, and `extent`; and each plane's `offset` and `rowPitch`.

Before creating the image, the application first verifies that the physical device supports the received creation parameters by querying `vkGetPhysicalDeviceFormatProperties2` with `VkDrmFormatModifierPropertiesListEXT` and `vkGetPhysicalDeviceImageFormatProperties2` with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`. Then the application creates the image by chaining `VkImageDrmFormatModifierExplicitCreateInfoEXT` and `VkExternalMemoryImageCreateInfo` onto `VkImageCreateInfo`.

- **Export.** The application creates an image and allocates its memory. Then the application exports to *modifier*-aware consumers the image's memory handles; its creation parameters; its *modifier*; and the `offset`, `size`, and `rowPitch` of each *memory plane*.

In this pattern, the Vulkan device is the authority for the image; it is the allocator of the image's memory and the decider of the image's creation parameters. When choosing the image's creation parameters, the application usually chooses a tuple (`format`, `drmFormatModifier`) from the result of the negotiation described above. The negotiation's result often contains multiple tuples that share the same `format` but differ in their *modifier*. In this case, the application should defer the choice of the image's *modifier* to the Vulkan implementation by providing all such *modifiers* to `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`; and the implementation should choose from `pDrmFormatModifiers` the optimal *modifier* in consideration with the other image parameters.

The application creates the image by chaining `VkImageDrmFormatModifierListCreateInfoEXT` and `VkExternalMemoryImageCreateInfo` onto `VkImageCreateInfo`. The protocol and APIs by which the application will share the image with external consumers will likely determine the value of `VkExternalMemoryImageCreateInfo::handleTypes`. The implementation chooses for the image an optimal *modifier* from `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`. The application then queries the implementation-chosen *modifier* with `vkGetImageDrmFormatModifierPropertiesEXT`, and queries the memory layout of each plane with `vkGetImageSubresourceLayout`.

The application then allocates the image's memory with `VkMemoryAllocateInfo`, adding chained extension structures for external memory; binds it to the image; and exports the memory, for example, with `vkGetMemoryFdKHR`.

Finally, the application sends the image's creation parameters, its *modifier*, its per-plane memory layout, and the exported memory handle to the external consumers. The details of how the application transmits this information to external consumers is outside the scope of this specification.

## Prior Art

Extension `EGL_EXT_image_dma_buf_import`<sup>1</sup> introduced the ability to create an `EGLImage` by importing for each plane a `dma_buf`, offset, and row pitch.

Later, extension `EGL_EXT_image_dma_buf_import_modifiers`<sup>2</sup> introduced the ability to query which combination of formats and *modifiers* the implementation supports and to specify *modifiers* during creation of the `EGLImage`.

Extension `EGL_MESA_image_dma_buf_export`<sup>3</sup> is the inverse of `EGL_EXT_image_dma_buf_import_modifiers`.

The Linux kernel modesetting API (KMS), when configuring the display's framebuffer with `struct drm_mode_fb_cmd2`<sup>4</sup>, allows one to specify the framebuffer's *modifier* as well as a per-plane memory handle, offset, and row pitch.

GBM, a graphics buffer manager for Linux, allows creation of a `gbm_bo` (that is, a graphics *buffer object*) by importing data similar to that in `EGL_EXT_image_dma_buf_import_modifiers`<sup>1</sup>; and symmetrically allows exporting the same data from the `gbm_bo`. See the references to *modifier* and *plane* in `gbm.h`<sup>5</sup>.

## New Object Types

None.

## New Enum Constants

- Extending `VkResult`:
  - `VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT`
  - `VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT`
  - `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT`
  - `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT`
  - `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT`
- Extending `VkImageTiling`:
  - `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`
- Extending `VkImageAspectFlagBits`:
  - `VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT`
  - `VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT`
  - `VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT`

- `VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT`

## New Enums

None.

## New Structures

- Extends `VkFormatProperties2`:
  - `VkDrmFormatModifierPropertiesListEXT`
- Member of `VkDrmFormatModifierPropertiesListEXT`:
  - `VkDrmFormatModifierPropertiesEXT`
- Extends `VkPhysicalDeviceImageFormatInfo2`:
  - `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`
- Extends `VkImageCreateInfo`:
  - `VkImageDrmFormatModifierListCreateInfoEXT`
  - `VkImageDrmFormatModifierExplicitCreateInfoEXT`
- Parameter to `vkGetImageDrmFormatModifierPropertiesEXT`:
  - `VkImageDrmFormatModifierPropertiesEXT`

## New Functions

- `vkGetImageDrmFormatModifierPropertiesEXT`

## Issues

1) Should this extension define a single DRM format modifier per `VkImage`? Or define one per plane?

+

**RESOLVED:** There exists a single DRM format modifier per `VkImage`.

**DISCUSSION:** Prior art, such as `EGL_EXT_image_dma_buf_import_modifiers`<sup>2</sup>, `struct drm_mode_fb_cmd2`<sup>4</sup>, and `struct gbm_import_fd_modifier_data`<sup>5</sup>, allows defining one *modifier* per plane. However, developers of the GBM and kernel APIs concede it was a mistake. Beginning in Linux 4.10, the kernel requires that the application provide the same DRM format *modifier* for each plane. (See Linux commit [bae781b259269590109e8a4a8227331362b88212](#)). And GBM provides an entrypoint, `gbm_bo_get_modifier`, for querying the *modifier* of the image but does not provide one to query the modifier of individual planes.

2) When creating an image with `VkImageDrmFormatModifierExplicitCreateInfoEXT`, which is typically used when *importing* an image, should the application explicitly provide the size of each plane?

+

**RESOLVED:** No. The application **must** not provide the size. To enforce this, the API requires that `VkImageDrmFormatModifierExplicitCreateInfoEXT::pPlaneLayouts::size` **must** be 0.

**DISCUSSION:** Prior art, such as `EGL_EXT_image_dma_buf_import_modifiers`<sup>2</sup>, `struct drm_mode_fb_cmd2`<sup>4</sup>, and `struct gbm_import_fd_modifier_data`<sup>5</sup>, omits from the API the size of each plane. Instead, the APIs infer each plane's size from the import parameters, which include the image's pixel format and a `dma_buf`, offset, and row pitch for each plane.

However, Vulkan differs from EGL and GBM with regards to image creation in the following ways:

#### *Differences in Image Creation*

- **Undedicated allocation by default.** When importing or exporting a set of `dma_bufs` as an `EGLImage` or `gbm_bo`, common practice mandates that each `dma_buf`'s memory be dedicated (in the sense of `VK_KHR_dedicated_allocation`) to the image (though not necessarily dedicated to a single plane). In particular, neither the GBM documentation nor the EGL extension specifications explicitly state this requirement, but in light of common practice this is likely due to under-specification rather than intentional omission. In contrast, `VK_EXT_image_drm_format_modifier` permits, but does not require, the implementation to require dedicated allocations for images created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`.
- **Separation of image creation and memory allocation.** When importing a set of `dma_bufs` as an `EGLImage` or `gbm_bo`, EGL and GBM create the image resource and bind it to memory (the `dma_bufs`) simultaneously. This allows EGL and GBM to query each `dma_buf`'s size during image creation. In Vulkan, image creation and memory allocation are independent unless a dedicated allocation is used (as in `VK_KHR_dedicated_allocation`). Therefore, without requiring dedicated allocation, Vulkan cannot query the size of each `dma_buf` (or other external handle) when calculating the image's memory layout. Even if dedication allocation were required, Vulkan cannot calculate the image's memory layout until after the image is bound to its `dma_ufs`.

The above differences complicate the potential inference of plane size in Vulkan. Consider the following problematic cases:

#### *Problematic Plane Size Calculations*

- **Padding.** Some plane of the image may require implementation-dependent padding.
- **Metadata.** For some *modifiers*, the image may have a metadata plane which requires a non-trivial calculation to determine its size.
- **Mipmapped, array, and 3D images.** The implementation may support `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` for images whose `mipLevels`, `arrayLayers`, or `depth` is greater than 1. For such images with certain *modifiers*, the calculation of each plane's size may be non-trivial.

However, an application-provided plane size solves none of the above problems.

For simplicity, consider an external image with a single memory plane. The implementation is obviously capable calculating the image's size when its tiling is `VK_IMAGE_TILING_OPTIMAL`. Likewise, any reasonable implementation is capable of calculating the image's size when its tiling uses a supported *modifier*.

Suppose that the external image's size is smaller than the implementation-calculated size. If the application provided the external image's size to `vkCreateImage`, the implementation would observe the mismatched size and recognize its inability to comprehend the external image's layout (unless the implementation used the application-provided size to select a refinement of the tiling layout indicated by the *modifier*, which is strongly discouraged). The implementation would observe the conflict, and reject image creation with `VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT`. On the other hand, if the application did not provide the external image's size to `vkCreateImage`, then the application would observe after calling `vkGetImageMemoryRequirements` that the external image's size is less than the size required by the implementation. The application would observe the conflict and refuse to bind the `VkImage` to the external memory. In both cases, the result is explicit failure.

Suppose that the external image's size is larger than the implementation-calculated size. If the application provided the external image's size to `vkCreateImage`, for reasons similar to above the implementation would observe the mismatched size and recognize its inability to comprehend the image data residing in the extra size. The implementation, however, must assume that image data resides in the entire size provided by the application. The implementation would observe the conflict and reject image creation with `VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT`. On the other hand, if the application did not provide the external image's size to `vkCreateImage`, then the application would observe after calling `vkGetImageMemoryRequirements` that the external image's size is larger than the implementation-usable size. The application would observe the conflict and refuse to bind the `VkImage` to the external memory. In both cases, the result is explicit failure.

Therefore, an application-provided size provides no benefit, and this extension should not require it. This decision renders `VkSubresourceLayout::size` an unused field during image creation, and thus introduces a risk that implementations may require applications to submit sideband creation parameters in the unused field. To prevent implementations from relying on sideband data, this extension *requires* the application to set `size` to 0.

## References

1. [EGL\\_EXT\\_image\\_dma\\_buf\\_import](#)
2. [EGL\\_EXT\\_image\\_dma\\_buf\\_import\\_modifiers](#)
3. [EGL\\_MESA\\_image\\_dma\\_buf\\_export](#)
4. [struct drm\\_mode\\_fb\\_cmd2](#)
5. [gbm.h](#)

## Version History

- Revision 1, 2018-08-29 (Chad Versace)
  - First stable revision

## VK\_EXT\_index\_type\_uint8

### Name String

`VK_EXT_index_type_uint8`

## **Extension Type**

Device extension

## **Registered Extension Number**

266

## **Revision**

1

## **Extension and Version Dependencies**

- Requires Vulkan 1.0

## **Contact**

- Piers Daniell [Opdaniell-nv](#)

## **Last Modified Date**

2019-05-02

## **IP Status**

No known IP claims.

## **Contributors**

- Jeff Bolz, NVIDIA

This extension allows `uint8_t` indices to be used with `vkCmdBindIndexBuffer`.

## **New Object Types**

None

## **New Enum Constants**

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT`
- Extending `VkIndexType`:
  - `VK_INDEX_TYPE_UINT8_EXT`

## **New Enums**

None

## **New Structures**

- `VkPhysicalDeviceIndexTypeUInt8FeaturesEXT`

## **New Functions**

None

## New Built-In Variables

None

## New SPIR-V Capabilities

None

## Issues

None

## Version History

- Revision 1, 2019-05-02 (Piers Daniell)
  - Internal revisions

## VK\_EXT\_inline\_uniform\_block

### Name String

`VK_EXT_inline_uniform_block`

### Extension Type

Device extension

### Registered Extension Number

139

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_maintenance1`

### Contact

- Daniel Rakos [Qaqnuep](#)

### Last Modified Date

2018-08-01

### IP Status

No known IP claims.

### Contributors

- Daniel Rakos, AMD

- Jeff Bolz, NVIDIA
- Slawomir Grajewski, Intel
- Neil Henning, Codeplay

This extension introduces the ability to back uniform blocks directly with descriptor sets by storing inline uniform data within descriptor pool storage. Compared to push constants this new construct allows uniform data to be reused across multiple disjoint sets of draw or dispatch commands and **may** enable uniform data to be accessed with less indirections compared to uniforms backed by buffer memory.

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT`
  - `VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO_EXT`
- Extending [VkDescriptorType](#):
  - `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`

## New Enums

None

## New Structures

- [VkPhysicalDeviceInlineUniformBlockFeaturesEXT](#)
- [VkPhysicalDeviceInlineUniformBlockPropertiesEXT](#)
- [VkWriteDescriptorSetInlineUniformBlockEXT](#)
- [VkDescriptorPoolInlineUniformBlockCreateInfoEXT](#)

## New Functions

None

## New Built-In Variables

None

## Issues

- 1) Do we need a new storage class for inline uniform blocks vs uniform blocks?

**RESOLVED:** No. The `Uniform` storage class is used to allow the same syntax used for both uniform buffers and inline uniform blocks.

2) Is the descriptor array index and array size expressed in terms of bytes or dwords for inline uniform block descriptors?

**RESOLVED:** In bytes, but both **must** be a multiple of 4, similar to how push constant ranges are specified. The `descriptorCount` of `VkDescriptorSetLayoutBinding` thus provides the total number of bytes a particular binding with an inline uniform block descriptor type can hold, while the `srcArrayElement`, `dstArrayElement`, and `descriptorCount` members of `VkWriteDescriptorSet`, `VkCopyDescriptorSet`, and `VkDescriptorUpdateTemplateEntry` (where applicable) specify the byte offset and number of bytes to write/copy to the binding's backing store. Additionally, the `stride` member of `VkDescriptorUpdateTemplateEntry` is ignored for inline uniform blocks and a default value of one is used, meaning that the data to update inline uniform block bindings with must be contiguous in memory.

3) What layout rules apply for uniform blocks corresponding to inline constants?

**RESOLVED:** They use the same layout rules as uniform buffers.

4) Do we need to add non-uniform indexing features/properties as introduced by `VK_EXT_descriptor_indexing` for inline uniform blocks?

**RESOLVED:** No, because inline uniform blocks are not allowed to be "arrayed". A single binding with an inline uniform block descriptor type corresponds to a single uniform block instance and the array indices inside that binding refer to individual offsets within the uniform block (see issue #2). However, this extension does introduce new features/properties about the level of support for update-after-bind inline uniform blocks.

5) Is the `descriptorBindingVariableDescriptorCount` feature introduced by `VK_EXT_descriptor_indexing` supported for inline uniform blocks?

**RESOLVED:** Yes, as long as other inline uniform block specific limits are respected.

6) Do the robustness guarantees of `robustBufferAccess` apply to inline uniform block accesses?

**RESOLVED:** No, similarly to push constants, as they are not backed by buffer memory like uniform buffers.

## Version History

- Revision 1, 2018-08-01 (Daniel Rakos)
  - Internal revisions

## `VK_EXT_line_rasterization`

### Name String

`VK_EXT_line_rasterization`

### Extension Type

Device extension

## Registered Extension Number

260

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Jeff Bolz [@jeffbolznv](#)

## Last Modified Date

2019-05-09

## IP Status

No known IP claims.

## Contributors

- Jeff Bolz, NVIDIA
- Allen Jensen, NVIDIA
- Jason Ekstrand, Intel

This extension adds some line rasterization features that are commonly used in CAD applications and supported in other APIs like OpenGL. Bresenham-style line rasterization is supported, smooth rectangular lines (coverage to alpha) are supported, and stippled lines are supported for all three line rasterization modes.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT`
- Extending [VkDynamicState](#):
  - `VK_DYNAMIC_STATE_LINE_STIPPLE_EXT`

## New Enums

- [VkLineRasterizationModeEXT](#)

## New Structures

- [VkPhysicalDeviceLineRasterizationFeaturesEXT](#)

- [VkPipelineRasterizationLineStateCreateInfoEXT](#)

## New Functions

- [vkCmdSetLineStippleEXT](#)

## Issues

- (1) Do we need to support Bresenham-style and smooth lines with more than one rasterization sample? i.e. the equivalent of `glDisable(GL_MULTISAMPLE)` in OpenGL when the framebuffer has more than one sample?

RESOLVED: Yes.

For simplicity, Bresenham line rasterization carries forward a few restrictions from OpenGL, such as not supporting per-sample shading, alpha to coverage, or alpha to one.

## Version History

- Revision 1, 2019-05-09 (Jeff Bolz)
  - Initial draft

## VK\_EXT\_memory\_budget

### Name String

`VK_EXT_memory_budget`

### Extension Type

Device extension

### Registered Extension Number

238

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Jeff Bolz [@jeffbolznv](#)

### Last Modified Date

2018-10-08

## Contributors

- Jeff Bolz, NVIDIA
- Jeff Juliano, NVIDIA

This extension adds support for querying the amount of memory used and the total memory budget for a memory heap. The values returned by this query are implementation-dependent and can depend on a variety of factors including operating system and system load.

The `heapBudget` values can be used as a guideline for how much total memory from each heap the process can use at any given time, before allocations may start failing or causing performance degradation. The values may change based on other activity in the system that is outside the scope and control of the Vulkan implementation.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceMemoryBudgetPropertiesEXT`

## New Functions

None.

## Version History

- Revision 1, 2018-10-08 (Jeff Bolz)
  - Initial revision

## `VK_EXT_memory_priority`

### Name String

`VK_EXT_memory_priority`

### Extension Type

Device extension

### Registered Extension Number

239

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Jeff Bolz [@jeffbolznv](#)

## Last Modified Date

2018-10-08

## Contributors

- Jeff Bolz, NVIDIA
- Jeff Juliano, NVIDIA

This extension adds a `priority` value specified at memory allocation time. On some systems with both device-local and non-device-local memory heaps, the implementation may transparently move memory from one heap to another when a heap becomes full (for example, when the total memory used across all processes exceeds the size of the heap). In such a case, this priority value may be used to determine which allocations are more likely to remain in device-local memory.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PRIORITY_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_MEMORY_PRIORITY_ALLOCATE_INFO_EXT`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceMemoryPriorityFeaturesEXT](#)
- [VkMemoryPriorityAllocateInfoEXT](#)

## New Functions

None.

## Version History

- Revision 1, 2018-10-08 (Jeff Bolz)
  - Initial revision

## [VK\\_EXT\\_metal\\_surface](#)

**Name String**

`VK_EXT_metal_surface`

**Extension Type**

Instance extension

**Registered Extension Number**

218

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

**Contact**

- Dzmitry Malyshau [Okvark](#)

**Last Modified Date**

2018-10-01

**IP Status**

No known IP claims.

**Contributors**

- Dzmitry Malyshau, Mozilla Corp.

The `VK_EXT_metal_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) from `CAMetalLayer`, which is the native rendering surface of Apple's Metal framework.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_METAL_SURFACE_CREATE_INFO_EXT`

## New Enums

None.

## New Structures

- `VkMetalSurfaceCreateInfoEXT`

## New Functions

- [vkCreateMetalSurfaceEXT](#)

## Issues

None.

## Version History

- Revision 1, 2018-10-01 (Dzmitry Malyshau)
  - Initial version

## `VK_EXT_pci_bus_info`

### Name String

`VK_EXT_pci_bus_info`

### Extension Type

Device extension

### Registered Extension Number

213

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Matthaeus G. Chajdas [Panteru](#)

### Last Modified Date

2018-12-10

### IP Status

No known IP claims.

### Contributors

- Matthaeus G. Chajdas, AMD
- Daniel Rakos, AMD

This extension adds a new query to obtain PCI bus information about a physical device.

Not all physical devices have PCI bus information, either due to the device not being connected to the system through a PCI interface or due to platform specific restrictions and policies. Thus this

extension is only expected to be supported by physical devices which can provide the information.

As a consequence, applications should always check for the presence of the extension string for each individual physical device for which they intend to issue the new query for and should not have any assumptions about the availability of the extension on any given platform.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PCI_BUS_INFO_PROPERTIES_EXT`

## New Enums

None.

## New Structures

- [VkPhysicalDevicePCIBusInfoPropertiesEXT](#)

## New Functions

None.

## Issues

None.

## Examples

None.

## Version History

- Revision 2, 2018-12-10 (Daniel Rakos)
  - Changed all members of the new structure to have the `uint32_t` type
- Revision 1, 2018-10-11 (Daniel Rakos)
  - Initial revision

## `VK_EXT_pipeline_creation_feedback`

### Name String

`VK_EXT_pipeline_creation_feedback`

## Extension Type

Device extension

## Registered Extension Number

193

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Jean-Francois Roy [Qjfroy](#)

## Last Modified Date

2019-03-12

## IP Status

No known IP claims.

## Contributors

- Jean-Francois Roy, Google
- Hai Nguyen, Google
- Andrew Ellem, Google
- Bob Fraser, Google
- Sujeevan Rajayogam, Google
- Jan-Harald Fredriksen, ARM
- Jeff Leger, Qualcomm Technologies, Inc.
- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA
- Neil Henning, AMD

This extension adds a mechanism to provide feedback to an application about pipeline creation, with the specific goal of allowing a feedback loop between build systems and in-the-field application executions to ensure effective pipeline caches are shipped to customers.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):

- `VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO_EXT`

## New Enums

- `VkPipelineCreationFeedbackFlagBitsEXT`

## New Structures

- `VkPipelineCreationFeedbackCreateInfoEXT`
- `VkPipelineCreationFeedbackEXT`

## New Functions

None.

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2019-03-12 (Jean-Francois Roy)
  - Initial revision

## `VK_EXT_post_depth_coverage`

### Name String

`VK_EXT_post_depth_coverage`

### Extension Type

Device extension

### Registered Extension Number

156

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Daniel Koch [@dgkoch](#)

## Last Modified Date

2017-07-17

## Interactions and External Dependencies

- This extension requires the `SPV_KHR_post_depth_coverage` SPIR-V extension.
- This extension requires `GL_ARB_post_depth_coverage` or `GL_EXT_post_depth_coverage` for GLSL-based source languages.

## Contributors

- Jeff Bolz, NVIDIA

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_KHR_post_depth_coverage`

which allows the fragment shader to control whether values in the `SampleMask` built-in input variable reflect the coverage after the `early per-fragment` depth and stencil tests are applied.

This extension adds a new `PostDepthCoverage` execution mode under the `SampleMaskPostDepthCoverage` capability. When this mode is specified along with `EarlyFragmentTests`, the value of an input variable decorated with the `SampleMask` built-in reflects the coverage after the `early fragment tests` are applied. Otherwise, it reflects the coverage before the depth and stencil tests.

When using GLSL source-based shading languages, the `post_depth_coverage` layout qualifier from `GL_ARB_post_depth_coverage` or `GL_EXT_post_depth_coverage` maps to the `PostDepthCoverage` execution mode.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New Built-In Variables

None.

## New Variable Decoration

None.

## New SPIR-V Capabilities

- [SampleMaskPostDepthCoverage](#)

## Issues

None yet.

## Version History

- Revision 1, 2017-07-17 (Daniel Koch)
  - Internal revisions

## VK\_EXT\_queue\_family\_foreign

### Name String

`VK_EXT_queue_family_foreign`

### Extension Type

Device extension

### Registered Extension Number

127

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_memory](#)

### Contact

- Chad Versace [@chadversary](#)

### Last Modified Date

2017-11-01

### IP Status

No known IP claims.

## Contributors

- Chad Versace, Google
- James Jones, NVIDIA
- Jason Ekstrand, Intel
- Jesse Hall, Google
- Daniel Rakos, AMD
- Ray Smith, ARM

This extension defines a special queue family, `VK_QUEUE_FAMILY_FOREIGN_EXT`, which can be used to transfer ownership of resources backed by external memory to foreign, external queues. This is similar to `VK_QUEUE_FAMILY_EXTERNAL_KHR`, defined in `VK_KHR_external_memory`. The key differences between the two are:

- The queues represented by `VK_QUEUE_FAMILY_EXTERNAL_KHR` must share the same physical device and the same driver version as the current `VkInstance`. `VK_QUEUE_FAMILY_FOREIGN_EXT` has no such restrictions. It can represent devices and drivers from other vendors, and can even represent non-Vulkan-capable devices.
- All resources backed by external memory support `VK_QUEUE_FAMILY_EXTERNAL_KHR`. Support for `VK_QUEUE_FAMILY_FOREIGN_EXT` is more restrictive.
- Applications should expect transitions to/from `VK_QUEUE_FAMILY_FOREIGN_EXT` to be more expensive than transitions to/from `VK_QUEUE_FAMILY_EXTERNAL_KHR`.

## New Enum Constants

- Special constants:
  - `VK_QUEUE_FAMILY_FOREIGN_EXT`

## Version History

- Revision 1, 2017-11-01 (Chad Versace)
  - Squashed internal revisions

## `VK_EXT_sample_locations`

### Name String

`VK_EXT_sample_locations`

### Extension Type

Device extension

### Registered Extension Number

144

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Daniel Rakos [@drakos-amd](#)

## Last Modified Date

2017-08-02

## Contributors

- Mais Alnasser, AMD
- Matthaeus G. Chajdas, AMD
- Maciej Jesionowski, AMD
- Daniel Rakos, AMD
- Slawomir Grajewski, Intel
- Jeff Bolz, NVIDIA
- Bill Licea-Kane, Qualcomm

This extension allows an application to modify the locations of samples within a pixel used in rasterization. Additionally, it allows applications to specify different sample locations for each pixel in a group of adjacent pixels, which **can** increase antialiasing quality (particularly if a custom resolve shader is used that takes advantage of these different locations).

It is common for implementations to optimize the storage of depth values by storing values that **can** be used to reconstruct depth at each sample location, rather than storing separate depth values for each sample. For example, the depth values from a single triangle **may** be represented using plane equations. When the depth value for a sample is needed, it is automatically evaluated at the sample location. Modifying the sample locations causes the reconstruction to no longer evaluate the same depth values as when the samples were originally generated, thus the depth aspect of a depth/stencil attachment **must** be cleared before rendering to it using different sample locations.

Some implementations **may** need to evaluate depth image values while performing image layout transitions. To accommodate this, instances of the [VkSampleLocationsInfoEXT](#) structure **can** be specified for each situation where an explicit or automatic layout transition has to take place. [VkSampleLocationsInfoEXT](#) **can** be chained from [VkImageMemoryBarrier](#) structures to provide sample locations for layout transitions performed by [vkCmdWaitEvents](#) and [vkCmdPipelineBarrier](#) calls, and [VkRenderPassSampleLocationsBeginInfoEXT](#) **can** be chained from [VkRenderPassBeginInfo](#) to provide sample locations for layout transitions performed implicitly by a render pass instance.

## New Object Types

None.

## New Enum Constants

- Extending [VkImageCreateFlagBits](#):
  - `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT`
- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT`
  - `VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT`
- Extending [VkDynamicState](#):
  - `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`

## New Enums

None.

## New Structures

- [VkSampleLocationEXT](#)
- [VkSampleLocationsInfoEXT](#)
- [VkAttachmentSampleLocationsEXT](#)
- [VkSubpassSampleLocationsEXT](#)
- [VkRenderPassSampleLocationsBeginInfoEXT](#)
- [VkPipelineSampleLocationsStateCreateInfoEXT](#)
- [VkPhysicalDeviceSampleLocationsPropertiesEXT](#)
- [VkMultisamplePropertiesEXT](#)

## New Functions

- [vkCmdSetSampleLocationsEXT](#)
- [vkGetPhysicalDeviceMultisamplePropertiesEXT](#)

## Issues

None.

## Version History

- Revision 1, 2017-08-02 (Daniel Rakos)
  - Internal revisions

## **VK\_EXT\_sampler\_filter\_minmax**

### **Name String**

`VK_EXT_sampler_filter_minmax`

### **Extension Type**

Device extension

### **Registered Extension Number**

131

### **Revision**

2

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### **Contact**

- Jeff Bolz [@jeffbolznv](#)

### **Last Modified Date**

2017-05-19

### **IP Status**

No known IP claims.

### **Contributors**

- Jeff Bolz, NVIDIA
- Piers Daniell, NVIDIA

In unextended Vulkan, minification and magnification filters such as LINEAR allow sampled image lookups to return a filtered texel value produced by computing a weighted average of a collection of texels in the neighborhood of the texture coordinate provided.

This extension provides a new sampler parameter which allows applications to produce a filtered texel value by computing a component-wise minimum (MIN) or maximum (MAX) of the texels that would normally be averaged. The reduction mode is orthogonal to the minification and magnification filter parameters. The filter parameters are used to identify the set of texels used to produce a final filtered value; the reduction mode identifies how these texels are combined.

## **New Object Types**

None.

## **New Enum Constants**

- Extending [VkStructureType](#):

- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES_EXT`
- `VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO_EXT`
- Extending `VkFormatFeatureFlagBits`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT_EXT`

## New Enums

- `VkSamplerReductionModeEXT`

## New Structures

- `VkSamplerReductionModeCreateInfoEXT`
- `VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT`

## New Functions

None.

## New Built-In Variables

None.

## New SPIR-V Capabilities

None.

## Issues

None.

## Examples

None.

## Version History

- Revision 2, 2017-05-19 (Piers Daniell)
  - Renamed to EXT
- Revision 1, 2017-03-25 (Jeff Bolz)
  - Internal revisions

## `VK_EXT_scalar_block_layout`

### Name String

`VK_EXT_scalar_block_layout`

### Extension Type

Device extension

## Registered Extension Number

222

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Tobias Hector [@tobski](#)

## Last Modified Date

2018-11-14

## Contributors

- Jeff Bolz
- Jan-Harald Fredriksen
- Graeme Leese
- Jason Ekstrand
- John Kessenich

## Short Description

Enables C-like structure layout for SPIR-V blocks.

## Description

This extension modifies the alignment rules for uniform buffers, storage buffers and push constants, allowing non-scalar types to be aligned solely based on the size of their components, without additional requirements.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_SCALAR\\_BLOCK\\_LAYOUT\\_FEATURES\\_EXT](#)

## New Structures

- [VkPhysicalDeviceScalarBlockLayoutFeaturesEXT](#)

## Issues

None.

## Version History

- Revision 1, 2018-11-14 (Tobias Hector)
  - Initial draft

## VK\_EXT\_separate\_stencil\_usage

### Name String

`VK_EXT_separate_stencil_usage`

### Extension Type

Device extension

### Registered Extension Number

247

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Daniel Rakos [@drakos-amd](#)

### Last Modified Date

2018-11-08

### IP Status

No known IP claims.

### Contributors

- Daniel Rakos, AMD
- Jordan Logan, AMD

This extension allows specifying separate usage flags for the stencil aspect of images with a depth-stencil format at image creation time.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO_EXT`

## New Enums

None.

## New Structures

- [VkImageStencilUsageCreateInfoEXT](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2018-11-08 (Daniel Rakos)
  - Internal revisions.

## `VK_EXT_shader_demote_to_helper_invocation`

### Name String

`VK_EXT_shader_demote_to_helper_invocation`

### Extension Type

Device extension

### Registered Extension Number

277

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Jeff Bolz [@jeffbolznv](#)

### Last Modified Date

## IP Status

No known IP claims.

## Contributors

- Jeff Bolz, NVIDIA

This extension adds Vulkan support for the `SPV_EXT_demote_to_helper_invocation` SPIR-V extension. That SPIR-V extension provides a new instruction `OpDemoteToHelperInvocationEXT` allowing shaders to "demote" a fragment shader invocation to behave like a helper invocation for its duration. The demoted invocation will have no further side effects and will not output to the framebuffer, but remains active and can participate in computing derivatives and in subgroup operations. This is a better match for the "discard" instruction in HLSL.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT`

## New Functions

None.

## New SPIR-V Capability

- `DemoteToHelperInvocationEXT`

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2019-06-01 (Jeff Bolz)
  - Initial draft

## VK\_EXT\_shader\_stencil\_export

### Name String

`VK_EXT_shader_stencil_export`

### Extension Type

Device extension

### Registered Extension Number

141

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Dominik Witczak [@dominikwitczakam](#)

### Last Modified Date

2017-07-19

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Requires the `SPV_EXT_shader_stencil_export` SPIR-V extension.

### Contributors

- Dominik Witczak, AMD
- Daniel Rakos, AMD
- Rex Xu, AMD

This extension adds support for the SPIR-V extension `SPV_EXT_shader_stencil_export`, providing a mechanism whereby a shader may generate the stencil reference value per invocation. When stencil testing is enabled, this allows the test to be performed against the value generated in the shader.

## Version History

- Revision 1, 2017-07-19 (Dominik Witczak)

- Initial draft

## **VK\_EXT\_shader\_subgroup\_ballot**

### **Name String**

`VK_EXT_shader_subgroup_ballot`

### **Extension Type**

Device extension

### **Registered Extension Number**

65

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Contact**

- Daniel Koch [Qdgkoch](#)

### **Last Modified Date**

2016-11-28

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

- This extension requires the `SPV_KHR_shader_ballot` SPIR-V extension.
- This extension requires the `GL_ARB_shader_ballot` extension for GLSL source languages.

### **Contributors**

- Jeff Bolz, NVIDIA
- Neil Henning, Codeplay
- Daniel Koch, NVIDIA Corporation

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_KHR_shader_ballot`

This extension provides the ability for a group of invocations, which execute in parallel, to do limited forms of cross-invocation communication via a group broadcast of a invocation value, or broadcast of a bitarray representing a predicate value from each invocation in the group.

This extension provides access to a number of additional built-in shader variables in Vulkan:

- `SubgroupEqMaskKHR`, which contains the subgroup mask of the current subgroup invocation,

- `SubgroupGeMaskKHR`, which contains the subgroup mask of the invocations greater than or equal to the current invocation,
- `SubgroupGtMaskKHR`, which contains the subgroup mask of the invocations greater than the current invocation,
- `SubgroupLeMaskKHR`, which contains the subgroup mask of the invocations less than or equal to the current invocation,
- `SubgroupLtMaskKHR`, which contains the subgroup mask of the invocations less than the current invocation,
- `SubgroupLocalInvocationId`, which contains the index of an invocation within a subgroup, and
- `SubgroupSize`, which contains the maximum number of invocations in a subgroup.

Additionally, this extension provides access to the new SPIR-V instructions:

- `OpSubgroupBallotKHR`,
- `OpSubgroupFirstInvocationKHR`, and
- `OpSubgroupReadInvocationKHR`,

When using GLSL source-based shader languages, the following variables and shader functions from GL\_ARB\_shader\_ballot can map to these SPIR-V built-in decorations and instructions:

- `in uint64_t gl_SubGroupEqMaskARB;` → `SubgroupEqMaskKHR`,
- `in uint64_t gl_SubGroupGeMaskARB;` → `SubgroupGeMaskKHR`,
- `in uint64_t gl_SubGroupGtMaskARB;` → `SubgroupGtMaskKHR`,
- `in uint64_t gl_SubGroupLeMaskARB;` → `SubgroupLeMaskKHR`,
- `in uint64_t gl_SubGroupLtMaskARB;` → `SubgroupLtMaskKHR`,
- `in uint gl_SubGroupInvocationARB;` → `SubgroupLocalInvocationId`,
- `uniform uint gl_SubGroupSizeARB;` → `SubgroupSize`,
- `ballotARB()` → `OpSubgroupBallotKHR`,
- `readFirstInvocationARB()` → `OpSubgroupFirstInvocationKHR`, and
- `readInvocationARB()` → `OpSubgroupReadInvocationKHR`.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New Built-In Variables

- `SubgroupEqMaskKHR`
- `SubgroupGeMaskKHR`
- `SubgroupGtMaskKHR`
- `SubgroupLeMaskKHR`
- `SubgroupLtMaskKHR`
- `SubgroupLocalInvocationId`
- `SubgroupSize`

## New SPIR-V Capabilities

- `SubgroupBallotKHR`

## Issues

None.

## Version History

- Revision 1, 2016-11-28 (Daniel Koch)
  - Initial draft

## `VK_EXT_shader_subgroup_vote`

### Name String

`VK_EXT_shader_subgroup_vote`

### Extension Type

Device extension

### Registered Extension Number

66

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Daniel Koch [@dgkoch](#)

## Last Modified Date

2016-11-28

## IP Status

No known IP claims.

## Interactions and External Dependencies

- This extension requires the [SPV\\_KHR\\_subgroup\\_vote](#) SPIR-V extension.
- This extension requires the [GL\\_ARB\\_shader\\_group\\_vote](#) extension for GLSL source languages.

## Contributors

- Neil Henning, Codeplay
- Daniel Koch, NVIDIA Corporation

This extension adds support for the following SPIR-V extension in Vulkan:

- [SPV\\_KHR\\_subgroup\\_vote](#)

This extension provides new SPIR-V instructions:

- [OpSubgroupAllKHR](#),
- [OpSubgroupAnyKHR](#), and
- [OpSubgroupAllEqualKHR](#).

to compute the composite of a set of boolean conditions across a group of shader invocations that are running concurrently (a *subgroup*). These composite results may be used to execute shaders more efficiently on a [VkPhysicalDevice](#).

When using GLSL source-based shader languages, the following shader functions from [GL\\_ARB\\_shader\\_group\\_vote](#) can map to these SPIR-V instructions:

- [anyInvocationARB\(\)](#) → [OpSubgroupAnyKHR](#),
- [allInvocationsARB\(\)](#) → [OpSubgroupAllKHR](#), and
- [allInvocationsEqualARB\(\)](#) → [OpSubgroupAllEqualKHR](#).

The subgroup across which the boolean conditions are evaluated is implementation-dependent, and this extension provides no guarantee over how individual shader invocations are assigned to subgroups. In particular, a subgroup has no necessary relationship with the compute shader *local workgroup*—any pair of shader invocations in a compute local workgroup may execute in different subgroups as used by these instructions.

Compute shaders operate on an explicitly specified group of threads (a local workgroup), but many implementations will also group non-compute shader invocations and execute them concurrently. When executing code like

```
if (condition) {
    result = do_fast_path();
} else {
    result = do_general_path();
}
```

where `condition` diverges between invocations, an implementation might first execute `do_fast_path()` for the invocations where `condition` is true and leave the other invocations dormant. Once `do_fast_path()` returns, it might call `do_general_path()` for invocations where `condition` is `false` and leave the other invocations dormant. In this case, the shader executes **both** the fast and the general path and might be better off just using the general path for all invocations.

This extension provides the ability to avoid divergent execution by evaluating a condition across an entire subgroup using code like:

```
if (allInvocationsARB(condition)) {
    result = do_fast_path();
} else {
    result = do_general_path();
}
```

The built-in function `allInvocationsARB()` will return the same value for all invocations in the group, so the group will either execute `do_fast_path()` or `do_general_path()`, but never both. For example, shader code might want to evaluate a complex function iteratively by starting with an approximation of the result and then refining the approximation. Some input values may require a small number of iterations to generate an accurate result (`do_fast_path`) while others require a larger number (`do_general_path`). In another example, shader code might want to evaluate a complex function (`do_general_path`) that can be greatly simplified when assuming a specific value for one of its inputs (`do_fast_path`).

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New Built-In Variables

None.

## New SPIR-V Capabilities

- [SubgroupVoteKHR](#)

## Issues

None.

## Version History

- Revision 1, 2016-11-28 (Daniel Koch)
  - Initial draft

## VK\_EXT\_shader\_viewport\_index\_layer

### Name String

`VK_EXT_shader_viewport_index_layer`

### Extension Type

Device extension

### Registered Extension Number

163

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Daniel Koch [@dgkoch](#)

### Last Modified Date

2017-08-08

### Interactions and External Dependencies

- This extension requires the `SPV_EXT_shader_viewport_index_layer` SPIR-V extension.
- This extension requires the `GL_ARB_shader_viewport_layer_array`, `GL_AMD_vertex_shader_layer`, `GL_AMD_vertex_shader_viewport_index`, or `GL_NV_viewport_array2` extensions for GLSL source

languages.

- This extension requires the `multiViewport` feature.
- This extension interacts with the `tessellationShader` feature.

## Contributors

- Piers Daniell, NVIDIA
- Jeff Bolz, NVIDIA
- Jan-Harald Fredriksen, ARM
- Daniel Rakos, AMD
- Slawomir Grajewski, Intel

This extension adds support for the `ShaderViewportIndexLayerEXT` capability from the `SPV_EXT_shader_viewport_index_layer` extension in Vulkan.

This extension allows variables decorated with the `Layer` and `ViewportIndex` built-ins to be exported from vertex or tessellation shaders, using the `ShaderViewportIndexLayerEXT` capability.

When using GLSL source-based shading languages, the `gl_ViewportIndex` and `gl_Layer` built-in variables map to the SPIR-V `ViewportIndex` and `Layer` built-in decorations, respectively. Behaviour of these variables is extended as described in the `GL_ARB_shader_viewport_layer_array` (or the precursor `GL_AMD_vertex_shader_layer`, `GL_AMD_vertex_shader_viewport_index`, and `GL_NV_viewport_array2` extensions).

### Note



The `ShaderViewportIndexLayerEXT` capability is equivalent to the `ShaderViewportIndexLayerNV` capability added by `VK_NV_viewport_array2`.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New or Modified Built-In Variables

- (modified) `Layer`
- (modified) `ViewportIndex`

## New Variable Decoration

None.

## New SPIR-V Capabilities

- `ShaderViewportIndexLayerEXT`

## Issues

None yet!

## Version History

- Revision 1, 2017-08-08 (Daniel Koch)
  - Internal drafts

## VK\_EXT\_subgroup\_size\_control

### Name String

`VK_EXT_subgroup_size_control`

### Extension Type

Device extension

### Registered Extension Number

226

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.1

### Contact

- Neil Henning [Osheredom](#)

### Last Modified Date

2019-03-05

### Contributors

- Jeff Bolz, NVIDIA
- Jason Ekstrand, Intel

- Sławek Grajewski, Intel
- Jesse Hall, Google
- Neil Henning, AMD
- Daniel Koch, NVIDIA
- Jeff Leger, Qualcomm
- Graeme Leese, Broadcom
- Allan MacKinnon, Google
- Mariusz Merecki, Intel
- Graham Wihlidal, Electronic Arts

## Short Description

Enables an implementation to control the subgroup size by allowing a varying subgroup size and also specifying a required subgroup size.

## Description

This extension extends the subgroup support in Vulkan 1.1 to allow an implementation to expose a varying subgroup size. Previously Vulkan exposed a single subgroup size per physical device, with the expectation that implementations will behave as if all subgroups have the same size. Some implementations **may** dispatch shaders with a varying subgroup size for different subgroups. As a result they could implicitly split a large subgroup into smaller subgroups or represent a small subgroup as a larger subgroup, some of whose invocations were inactive on launch.

To aid developers in understanding the performance characteristics of their programs, this extension exposes a minimum and maximum subgroup size that a physical device supports and a pipeline create flag to enable that pipeline to vary its subgroup size. If enabled, any [SubgroupSize](#) decorated variables in the SPIR-V shader modules provided to pipeline creation **may** vary between the [minimum](#) and [maximum](#) subgroup sizes.

An implementation is also optionally allowed to support specifying a required subgroup size for a given pipeline stage. Implementations advertise which [stages support a required subgroup size](#), and any pipeline of a supported stage can be passed a [VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT](#) structure to set the subgroup size for that shader stage of the pipeline. For compute shaders, this requires the developer to query the [maxComputeWorkgroupSubgroups](#) and ensure that:

$$s = \text{WorkGroupSize.x} * \text{WorkGroupSize.y} * \text{WorkgroupSize.z} \leq \text{SubgroupSize} * \text{maxComputeWorkgroupSubgroups}$$

Developers can also specify a new pipeline shader stage create flag that requires the implementation to have fully populated subgroups within local workgroups. This requires the workgroup size in the X dimension to be a multiple of the subgroup size.

## New Enum Constants

- Extending [VkStructureType](#):

- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES_EXT` (added in version 2)
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT`
- `VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO_EXT`
- Extending `VkPipelineShaderStageCreateFlagBits`:
  - `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT`
  - `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT`

## New Structures

- `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` (added in version 2)
- `VkPhysicalDeviceSubgroupSizeControlPropertiesEXT`
- `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT`

## Issues

None.

## Version History

- Revision 1, 2019-03-05 (Neil Henning)
  - Initial draft
- Revision 2, 2019-07-26 (Jason Ekstrand)
  - Add the missing `VkPhysicalDeviceSubgroupSizeControlFeaturesEXT` for querying subgroup size control features.

## `VK_EXT_swapchain_colorspace`

### Name String

`VK_EXT_swapchain_colorspace`

### Extension Type

Instance extension

### Registered Extension Number

105

### Revision

4

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

### Contact

- Courtney Goeltzenleuchter [Courtney-g](#)

## Last Modified Date

2019-04-26

## IP Status

No known IP claims.

## Contributors

- Courtney Goeltzenleuchter, Google

## New Enum Constants

- Extending `VkColorSpaceKHR`:
  - `VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT` - supports the Display-P3 color space and applies an sRGB-like transfer function.
  - `VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT` - supports the extended sRGB color space and applies a linear transfer function.
  - `VK_COLOR_SPACE_EXTENDED_SRGB_NONLINEAR_EXT` - supports the extended sRGB color space with an sRGB nonlinear transfer function.
  - `VK_COLOR_SPACE_DCI_P3_LINEAR_EXT` - supports the DCI-P3 color space and applies a linear OETF.
  - `VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT` - supports the DCI-P3 color space and applies the Gamma 2.6 OETF.
  - `VK_COLOR_SPACE_BT709_LINEAR_EXT` - supports the BT709 color space and applies a linear transfer function.
  - `VK_COLOR_SPACE_BT709_NONLINEAR_EXT` - supports the BT709 color space and applies the SMPTE 170M OETF.
  - `VK_COLOR_SPACE_BT2020_LINEAR_EXT` - supports the BT2020 color space and applies a linear OETF.
  - `VK_COLOR_SPACE_HDR10_ST2084_EXT` - supports HDR10 (BT2020 color space and applies the SMPTE ST2084 Perceptual Quantizer (PQ) OETF).
  - `VK_COLOR_SPACE_DOLBYVISION_EXT` - supports Dolby Vision (BT2020 color space, proprietary encoding, and applies the SMPTE ST2084 OETF).
  - `VK_COLOR_SPACE_HDR10_HLG_EXT` - supports HDR10 (BT2020 color space and applies the Hybrid Log Gamma (HLG) OETF).
  - `VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT` - supports the AdobeRGB color space and applies a linear OETF.
  - `VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT` - supports the AdobeRGB color space and applies the Gamma 2.2 OETF.
  - `VK_COLOR_SPACE_PASS_THROUGH_EXT` - color components used “as is”. Intended to allow application to supply data for color spaces not described here.

## Issues

1) Does the spec need to specify which kinds of image formats support the color spaces?

**RESOLVED:** Pixel format is independent of color space (though some color spaces really want / need floating point color components to be useful). Therefore, do not plan on documenting what formats support which colorspace. An application can call [vkGetPhysicalDeviceSurfaceFormatsKHR](#) to query what a particular implementation supports.

2) How does application determine if HW supports appropriate transfer function for a colorspace?

**RESOLVED:** Extension indicates that implementation **must** not do the OETF encoding if it is not sRGB. That responsibility falls to the application shaders. Any other native OETF / EOTF functions supported by an implementation can be described by separate extension.

## Version History

- Revision 1, 2016-12-27 (Courtney Goeltzenleuchter)
  - Initial version
- Revision 2, 2017-01-19 (Courtney Goeltzenleuchter)
  - Add pass through and multiple options for BT2020.
  - Clean up some issues with equations not displaying properly.
- Revision 3, 2017-06-23 (Courtney Goeltzenleuchter)
  - Add extended sRGB non-linear enum.
- Revision 4, 2019-04-26 (Graeme Leese)
  - Clarify colorspace transfer function usage.
  - Refer to normative definitions in the Data Format Specification.
  - Clarify DCI-P3 and Display P3 usage.

## VK\_EXT\_texel\_buffer\_alignment

### Name String

`VK_EXT_texel_buffer_alignment`

### Extension Type

Device extension

### Registered Extension Number

282

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

- Requires `VK_KHR_get_physical_device_properties2`

## Contact

- Jeff Bolz [@jeffbolznv](#)

## Last Modified Date

2019-06-06

## IP Status

No known IP claims.

## Interactions and External Dependencies

### Contributors

- Jeff Bolz, NVIDIA

This extension adds more expressive alignment requirements for uniform and storage texel buffers. Some implementations have single texel alignment requirements that can't be expressed via `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT`
- `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT`

## New Functions

None.

## Issues

## Version History

- Revision 1, 2019-06-06 (Jeff Bolz)
  - Initial draft

## VK\_EXT\_texture\_compression\_astc\_hdr

### Name String

`VK_EXT_texture_compression_astc_hdr`

### Extension Type

Device extension

### Registered Extension Number

67

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Jan-Harald Fredriksen [janharaldfredriksen-arm](#)

### Last Modified Date

2019-05-28

### IP Status

No known issues.

### Contributors

- Jan-Harald Fredriksen, Arm

This extension adds support for textures compressed using the Adaptive Scalable Texture Compression (ASTC) High Dynamic Range (HDR) profile.

When this extension is enabled, the HDR profile is supported for all ASTC formats listed in [ASTC Compressed Image Formats](#).

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT`
- Extending [VkFormat](#):
  - `VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT`

- `VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT`
- `VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT`

## New Functions

None.

## Issues

1) Should we add a feature or limit for this functionality?

Yes. It is consistent with the ASTC LDR support to add a feature like `textureCompressionASTC_HDR`.

The feature is strictly speaking redundant as long as this is just an extension; it would be sufficient to just enable the extension. But adding the feature is more forward-looking if wanted to make this an optional core feature in the future.

2) Should we introduce new format enums for HDR?

Yes. Vulkan 1.0 describes the ASTC format enums as `UNORM`, e.g. `VK_FORMAT_ASTC_4x4_UNORM_BLOCK`, so it's confusing to make these contain HDR data. Note that the OpenGL (ES) extensions did not make this distinction because a single ASTC HDR texture may contain both unorm and float blocks. Implementations **may** not be able to distinguish between LDR and HDR ASTC textures internally and just treat them as the same format, i.e. if this extension is supported then sampling from an `VK_FORMAT_ASTC_4x4_UNORM_BLOCK` image format **may** return HDR results. Applications **can** get predictable results by using the appropriate image format.

## Version History

- Revision 1, 2019-05-28 (Jan-Harald Fredriksen)

- Initial version

## **VK\_EXT\_transform\_feedback**

### **Name String**

`VK_EXT_transform_feedback`

### **Extension Type**

Device extension

### **Registered Extension Number**

29

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### **Contact**

- Piers Daniell [Opdaniell-nv](#)

### **Last Modified Data**

2018-10-09

### **Contributors**

- Baldur Karlsson, Valve
- Boris Zanin, Mobica
- Daniel Rakos, AMD
- Donald Scorgie, Imagination
- Henri Verbeet, CodeWeavers
- Jan-Harald Fredriksen, Arm
- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA
- Jesse Barker, Unity
- Jesse Hall, Google
- Pierre-Loup Griffais, Valve
- Philip Rebohle, DXVK
- Ruihao Zhang, Qualcomm
- Samuel Pitoiset, Valve
- Slawomir Grajewski, Intel

- Stu Smith, Imagination Technologies

This extension adds transform feedback to the Vulkan API by exposing the SPIR-V `TransformFeedback` and `GeometryStreams` capabilities to capture vertex, tessellation or geometry shader outputs to one or more buffers. It adds API functionality to bind transform feedback buffers to capture the primitives emitted by the graphics pipeline from SPIR-V outputs decorated for transform feedback. The transform feedback capture can be paused and resumed by way of storing and retrieving a byte counter. The captured data can be drawn again where the vertex count is derived from the byte counter without CPU intervention. If the implementation is capable, a vertex stream other than zero can be rasterized.

All these features are designed to match the full capabilities of OpenGL core transform feedback functionality and beyond. Many of the features are optional to allow base OpenGL ES GPUs to also implement this extension.

The primary purpose of the functionality exposed by this extension is to support translation layers from other 3D APIs. This functionality is not considered forward looking, and is not expected to be promoted to a KHR extension or to core Vulkan. Unless this is needed for translation, it is recommended that developers use alternative techniques of using the GPU to process and capture vertex data.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_STREAM_CREATE_INFO_EXT`
- Extending `VkQueryType`:
  - `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT`
- Extending `VkBufferUsageFlagBits`:
  - `VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT`
  - `VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT`
- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT`
  - `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT`
  - `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT`
- Extending `VkPipelineStageFlagBits`:
  - `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`

## New Enums

- `VkPipelineRasterizationStateStreamCreateFlagsEXT`

## New Structures

- Extending [VkPhysicalDeviceFeatures2](#):
  - [VkPhysicalDeviceTransformFeedbackFeaturesEXT](#)
- Extending [VkPhysicalDeviceProperties2](#):
  - [VkPhysicalDeviceTransformFeedbackPropertiesEXT](#)
- Extending [VkPipelineRasterizationStateCreateInfo](#)
  - [VkPipelineRasterizationStateStreamCreateInfoEXT](#)

## New Functions

- [vkCmdBindTransformFeedbackBuffersEXT](#)
- [vkCmdBeginTransformFeedbackEXT](#)
- [vkCmdEndTransformFeedbackEXT](#)
- [vkCmdBeginQueryIndexedEXT](#)
- [vkCmdEndQueryIndexedEXT](#)
- [vkCmdDrawIndirectByteCountEXT](#)

## Issues

1) Should we include pause/resume functionality?

**RESOLVED:** Yes, this is needed to ease layering other APIs which have this functionality. To pause use [vkCmdEndTransformFeedbackEXT](#) and provide valid buffer handles in the [pCounterBuffers](#) array and offsets in the [pCounterBufferOffsets](#) array for the implementation to save the resume points. Then to resume use [vkCmdBeginTransformFeedbackEXT](#) with the previous [pCounterBuffers](#) and [pCounterBufferOffsets](#) values. Between the pause and resume there needs to be a memory barrier for the counter buffers with a source access of [VK\\_ACCESS\\_TRANSFORM\\_FEEDBACK\\_COUNTER\\_WRITE\\_BIT\\_EXT](#) at pipeline stage [VK\\_PIPELINE\\_STAGE\\_TRANSFORM\\_FEEDBACK\\_BIT\\_EXT](#) to a destination access of [VK\\_ACCESS\\_TRANSFORM\\_FEEDBACK\\_COUNTER\\_READ\\_BIT\\_EXT](#) at pipeline stage [VK\\_PIPELINE\\_STAGE\\_DRAW\\_INDIRECT\\_BIT](#).

2) How does this interact with multiview?

**RESOLVED:** Transform feedback cannot be made active in a render pass with multiview enabled.

3) How should queries be done?

**RESOLVED:** There is a new query type [VK\\_QUERY\\_TYPE\\_TRANSFORM\\_FEEDBACK\\_STREAM\\_EXT](#). A query pool created with this type will capture 2 integers - numPrimitivesWritten and numPrimitivesNeeded - for the specified vertex stream output from the last vertex processing stage. The vertex stream output queried is zero by default, but can be specified with the new [vkCmdBeginQueryIndexedEXT](#) and [vkCmdEndQueryIndexedEXT](#) commands.

## Version History

- Revision 1, 2018-10-09 (Piers Daniell)
  - Internal revisions

## VK\_EXT\_validation\_cache

### Name String

`VK_EXT_validation_cache`

### Extension Type

Device extension

### Registered Extension Number

161

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Cort Stratton [@cdwfs](#)

### Last Modified Date

2017-08-29

### IP Status

No known IP claims.

### Contributors

- Cort Stratton, Google
- Chris Forbes, Google

This extension provides a mechanism for caching the results of potentially expensive internal validation operations across multiple runs of a Vulkan application. At the core is the `VkValidationCacheEXT` object type, which is managed similarly to the existing `VkPipelineCache`.

The new struct `VkShaderModuleValidationCacheCreateInfoEXT` can be included in the `pNext` chain at `vkCreateShaderModule` time. It contains a `VkValidationCacheEXT` to use when validating the `VkShaderModule`.

## New Object Types

- `VkValidationCacheEXT`

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_VALIDATION_CACHE_CREATE_INFO_EXT`
  - `VK_STRUCTURE_TYPE_SHADER_MODULE_VALIDATION_CACHE_CREATE_INFO_EXT`

## New Enums

- [VkValidationCacheHeaderVersionEXT](#)
- [VkValidationCacheCreateFlagsEXT](#)

## New Structures

- [VkValidationCacheCreateInfoEXT](#)
- [VkShaderModuleValidationCacheCreateInfoEXT](#)

## New Functions

- [vkCreateValidationCacheEXT](#)
- [vkDestroyValidationCacheEXT](#)
- [vkMergeValidationCachesEXT](#)
- [vkGetValidationCacheDataEXT](#)

## Issues

None.

## Version History

- Revision 1, 2017-08-29 (Cort Stratton)
  - Initial draft

## VK\_EXT\_validation\_features

### Name String

`VK_EXT_validation_features`

### Extension Type

Instance extension

### Registered Extension Number

248

### Revision

2

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Karl Schultz [karl-lunarg](#)

## Last Modified Date

2018-11-14

## IP Status

No known IP claims.

## Contributors

- Karl Schultz, LunarG
- Dave Houlton, LunarG
- Mark Lobodzinski, LunarG
- Camden Stocker, LunarG

This extension provides the `VkValidationFeaturesEXT` struct that can be included in the `pNext` chain of the `VkInstanceCreateInfo` structure passed as the `pCreateInfo` parameter of `vkCreateInstance`. The structure contains an array of `VkValidationFeatureEnableEXT` enum values that enable specific validation features that are disabled by default. The structure also contains an array of `VkValidationFeatureDisableEXT` enum values that disable specific validation layer features that are enabled by default.

### Note



The `VK_EXT_validation_features` extension subsumes all the functionality provided in the `VK_EXT_validation_flags` extension.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT`

## New Enums

- `VkValidationFeatureEnableEXT`
- `VkValidationFeatureDisableEXT`

## New Structures

- `VkValidationFeaturesEXT`

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2018-11-14 (Karl Schultz)
  - Initial revision
- Revision 2, 2019-08-06 (Mark Lobodzinski)
  - Add Best Practices enable

## VK\_EXT\_vertex\_attribute\_divisor

### Name String

`VK_EXT_vertex_attribute_divisor`

### Extension Type

Device extension

### Registered Extension Number

191

### Revision

3

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Vikram Kushwaha [@vkushwaha](#)

### Last Modified Date

2018-08-03

### IP Status

No known IP claims.

### Contributors

- Vikram Kushwaha, NVIDIA
- Jason Ekstrand, Intel

This extension allows instance-rate vertex attributes to be repeated for certain number of instances instead of advancing for every instance when instanced rendering is enabled.

## New Object Types

None.

## New Enum Constants

Extending [VkStructureType](#):

- [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_VERTEX\\_ATTRIBUTE\\_DIVISOR\\_PROPERTIES\\_EXT](#)
- [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_VERTEX\\_INPUT\\_DIVISOR\\_STATE\\_CREATE\\_INFO\\_EXT](#)
- [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_VERTEX\\_ATTRIBUTE\\_DIVISOR\\_FEATURES\\_EXT](#)

## New Enums

None.

## New Structures

- Extending [VkPipelineVertexInputStateCreateInfo](#):
  - [VkPipelineVertexInputDivisorStateCreateInfoEXT](#)
- [VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT](#)
- [VkVertexInputBindingDivisorDescriptionEXT](#)
- Extending [VkPhysicalDeviceFeatures2](#):
  - [VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT](#)

## New Functions

None.

## Issues

1) What is the effect of a non-zero value for `firstInstance`?

**RESOLVED:** The Vulkan API should follow the OpenGL convention and offset attribute fetching by `firstInstance` while computing vertex attribute offsets.

2) Should zero be an allowed divisor?

**RESOLVED:** Yes. A zero divisor means the vertex attribute is repeated for all instances.

## Examples

To create a vertex binding such that the first binding uses instanced rendering and the same attribute is used for every 4 draw instances, an application could use the following set of structures:

```

const VkVertexInputBindingDivisorDescriptionEXT divisorDesc =
{
    0,
    4
};

const VkPipelineVertexInputDivisorStateCreateInfoEXT divisorInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT, // sType
    NULL, // pNext
    1, // // vertexBindingDivisorCount
    &divisorDesc // // pVertexBindingDivisors
};

const VkVertexInputBindingDescription binding =
{
    0, // binding
    sizeof(Vertex), // stride
    VK_VERTEX_INPUT_RATE_INSTANCE // inputRate
};

const VkPipelineVertexInputStateCreateInfo viInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_CREATE_INFO, // sType
    &divisorInfo, // pNext
    ...
};
//...

```

## Version History

- Revision 1, 2017-12-04 (Vikram Kushwaha)
  - First Version
- Revision 2, 2018-07-16 (Jason Ekstrand)
  - Adjust the interaction between `divisor` and `firstInstance` to match the OpenGL convention.
  - Disallow divisors of zero.
- Revision 3, 2018-08-03 (Vikram Kushwaha)
  - Allow a zero divisor.
  - Add a physical device features structure to query/enable this feature.

## VK\_EXT\_ycbcr\_image\_arrays

**Name String**

`VK_EXT_ycbcr_image_arrays`

**Extension Type**

Device extension

**Registered Extension Number**

253

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_sampler_ycbcr_conversion`

**Contact**

- Piers Daniell [Opdaniell-nv](#)

**Last Modified Date**

2019-01-15

**Contributors**

- Piers Daniell, NVIDIA

This extension allows images of a format that requires `Y'CbCr` conversion to be created with multiple array layers, which is otherwise restricted.

**New Enum Constants**

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT`

**New Enums**

None.

**New Structures**

- `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT`

**New Functions**

None.

**Version History**

- Revision 1, 2019-01-15 (Piers Daniell)

- Initial revision

## **VK\_AMD\_buffer\_marker**

### **Name String**

`VK_AMD_buffer_marker`

### **Extension Type**

Device extension

### **Registered Extension Number**

180

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Contact**

- Daniel Rakos [@drakos-amd](#)

### **Last Modified Date**

2018-01-26

### **IP Status**

No known IP claims.

### **Contributors**

- Matthaeus G. Chajdas, AMD
- Jaakkko Konttinen, AMD
- Daniel Rakos, AMD

This extension adds a new operation to execute pipelined writes of small marker values into a `VkBuffer` object.

The primary purpose of these markers is to facilitate the development of debugging tools for tracking which pipelined command contributed to device loss.

### **New Object Types**

None.

### **New Enum Constants**

None.

## New Enums

None.

## New Structures

None.

## New Functions

- [vkCmdWriteBufferMarkerAMD](#)

## Examples

None.

## Version History

- Revision 1, 2018-01-26 (Jaakko Konttinen)
  - Initial revision

## VK\_AMD\_device\_coherent\_memory

### Name String

`VK_AMD_device_coherent_memory`

### Extension Type

Device extension

### Registered Extension Number

230

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Tobias Hector [@tobski](#)

### Last Modified Date

2019-02-04

### Contributors

- Ping Fu, AMD
- Timothy Lottes, AMD
- Tobias Hector, AMD

This extension adds the device coherent and device uncached memory types. Any device accesses to device coherent memory are automatically made visible to any other device access. Device uncached memory indicates to applications that caches are disabled for a particular memory type, which guarantees device coherence.

Device coherent and uncached memory are expected to have lower performance for general access than non-device coherent memory, but can be useful in certain scenarios; particularly so for debugging.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COHERENT_MEMORY_FEATURES_AMD`
- Extending `VkMemoryPropertyFlagBits`:
  - `VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD`
  - `VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD`

## Version History

- Revision 1, 2019-02-04 (Tobias Hector)
  - Initial revision

## **VK\_AMD\_display\_native\_hdr**

### Name String

`VK_AMD_display_native_hdr`

### Extension Type

Device extension

### Registered Extension Number

214

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_get_surface_capabilities2`
- Requires `VK_KHR_swapchain`

### Contact

- Matthaeus G. Chajdas [Qanteru](#)

### Last Modified Date

## IP Status

No known IP claims.

## Contributors

- Matthaeus G. Chajdas, AMD
- Aaron Hagan, AMD
- Aric Cyr, AMD
- Timothy Lottes, AMD
- Derrick Owens, AMD
- Daniel Rakos, AMD

This extension introduces the following display native HDR features to Vulkan:

- A new [VkColorSpaceKHR](#) enum for setting the native display colorspace. For example, this color space would be set by the swapchain to use the native color space in Freesync2 displays.
- Local dimming control

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_DISPLAY_NATIVE_HDR_SURFACE_CAPABILITIES_AMD`
  - `VK_STRUCTURE_TYPE_SWAPCHAIN_DISPLAY_NATIVE_HDR_CREATE_INFO_AMD`
- Extending [VkColorSpaceKHR](#):
  - `VK_COLOR_SPACE_DISPLAY_NATIVE_AMD`

## New Enums

None.

## New Structures

- [VkDisplayNativeHdrSurfaceCapabilitiesAMD](#)
- [VkSwapchainDisplayNativeHdrCreateInfoAMD](#)

## New Functions

- [vkSetLocalDimmingAMD](#)

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2018-12-18 (Daniel Rakos)
  - Initial revision

## VK\_AMD\_gcn\_shader

### Name String

`VK_AMD_gcn_shader`

### Extension Type

Device extension

### Registered Extension Number

26

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Dominik Witczak [@dominikwitczakamd](#)

### Last Modified Date

2016-05-30

### IP Status

No known IP claims.

### Contributors

- Dominik Witczak, AMD
- Daniel Rakos, AMD
- Rex Xu, AMD
- Graham Sellers, AMD

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_AMD_gcn_shader`

## Version History

- Revision 1, 2016-05-30 (Dominik Witczak)
  - Initial draft

## VK\_AMD\_memory\_overallocation\_behavior

### Name String

`VK_AMD_memory_overallocation_behavior`

### Extension Type

Device extension

### Registered Extension Number

190

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Martin Dinkov [@mdinkov](#)

### Last Modified Date

2018-09-19

### IP Status

No known IP claims.

### Contributors

- Martin Dinkov, AMD
- Matthaeus Chajdas, AMD
- Daniel Rakos, AMD
- Jon Campbell, AMD

This extension allows controlling whether explicit overallocation beyond the device memory heap sizes (reported by `VkPhysicalDeviceMemoryProperties`) is allowed or not. Overallocation may lead to performance loss and is not supported for all platforms.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_DEVICE_MEMORY_OVERALLOCATION_CREATE_INFO_AMD`

## New Enums

- [VkMemoryOverallocationBehaviorAMD](#)

## New Structures

- [VkDeviceMemoryOverallocationCreateInfoAMD](#)

## New Functions

None.

## Examples

None.

## Version History

- Revision 1, 2018-09-19 (Martin Dinkov)
  - Initial draft.

## **VK\_AMD\_mixed\_attachment\_samples**

### Name String

`VK_AMD_mixed_attachment_samples`

### Extension Type

Device extension

### Registered Extension Number

137

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Matthaeus G. Chajdas [Qanteru](#)

### Last Modified Date

2017-07-24

## Contributors

- Mais Alnasser, AMD
- Matthaeus G. Chajdas, AMD
- Maciej Jesionowski, AMD
- Daniel Rakos, AMD

This extension enables applications to use multisampled rendering with a depth/stencil sample count that is larger than the color sample count. Having a depth/stencil sample count larger than the color sample count allows maintaining geometry and coverage information at a higher sample rate than color information. All samples are depth/stencil tested, but only the first color sample count number of samples get a corresponding color output.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2017-07-24 (Daniel Rakos)
  - Internal revisions

## VK\_AMD\_pipeline\_compiler\_control

### Name String

`VK_AMD_pipeline_compiler_control`

### Extension Type

Device extension

## Registered Extension Number

184

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Matthaeus G. Chajdas [@anteru](#)

## Last Modified Date

2019-07-26

## IP Status

No known IP claims.

## Contributors

- Matthaeus G. Chajdas, AMD
- Daniel Rakos, AMD
- Maciej Jesionowski, AMD
- Tobias Hector, AMD

This extension introduces [VkPipelineCompilerControlCreateInfoAMD](#) structure that can be chained to a pipeline's create info to specify additional flags that affect pipeline compilation.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PIPELINE_COMPILER_CONTROL_CREATE_INFO_AMD`

## New Enums

- [VkPipelineCompilerControlFlagBitsAMD](#)

## New Structures

- [VkPipelineCompilerControlCreateInfoAMD](#)

## New Functions

None.

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2019-07-26 (Tobias Hector)
  - Initial revision.

## **VK\_AMD\_rasterization\_order**

### Name String

`VK_AMD_rasterization_order`

### Extension Type

Device extension

### Registered Extension Number

19

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Daniel Rakos [@drakos-amd](#)

### Last Modified Date

2016-04-25

### IP Status

No known IP claims.

### Contributors

- Matthaeus G. Chajdas, AMD
- Jaakko Konttinen, AMD
- Daniel Rakos, AMD

- Graham Sellers, AMD
- Dominik Witczak, AMD

This extension introduces the possibility for the application to control the order of primitive rasterization. In unextended Vulkan, the following stages are guaranteed to execute in *API order*:

- depth bounds test
- stencil test, stencil op, and stencil write
- depth test and depth write
- occlusion queries
- blending, logic op, and color write

This extension enables applications to opt into a relaxed, implementation defined primitive rasterization order that may allow better parallel processing of primitives and thus enabling higher primitive throughput. It is applicable in cases where the primitive rasterization order is known to not affect the output of the rendering or any differences caused by a different rasterization order are not a concern from the point of view of the application's purpose.

A few examples of cases when using the relaxed primitive rasterization order would not have an effect on the final rendering:

- If the primitives rendered are known to not overlap in framebuffer space.
- If depth testing is used with a comparison operator of `VK_COMPARE_OP_LESS`, `VK_COMPARE_OP_LESS_OR_EQUAL`, `VK_COMPARE_OP_GREATER`, or `VK_COMPARE_OP_GREATER_OR_EQUAL`, and the primitives rendered are known to not overlap in clip space.
- If depth testing is not used and blending is enabled for all attachments with a commutative blend operator.

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_RASTERIZATION_ORDER_AMD`

## New Enums

- [VkRasterizationOrderAMD](#)

## New Structures

- [VkPipelineRasterizationStateRasterizationOrderAMD](#)

## New Functions

None

## Issues

1) How is this extension useful to application developers?

**RESOLVED:** Allows them to increase primitive throughput for cases when strict API order rasterization is not important due to the nature of the content, the configuration used, or the requirements towards the output of the rendering.

2) How does this extension interact with content optimizations aiming to reduce overdraw by appropriately ordering the input primitives?

**RESOLVED:** While the relaxed rasterization order might somewhat limit the effectiveness of such content optimizations, most of the benefits of it are expected to be retained even when the relaxed rasterization order is used, so applications **should** still apply these optimizations even if they intend to use the extension.

3) Are there any guarantees about the primitive rasterization order when using the new relaxed mode?

**RESOLVED:** No. In this case the rasterization order is completely implementation dependent, but in practice it is expected to partially still follow the order of incoming primitives.

4) Does the new relaxed rasterization order have any adverse effect on repeatability and other invariance rules of the API?

**RESOLVED:** Yes, in the sense that it extends the list of exceptions when the repeatability requirement does not apply.

## Examples

None

## Issues

None

## Version History

- Revision 1, 2016-04-25 (Daniel Rakos)
  - Initial draft.

## VK\_AMD\_shader\_ballot

### Name String

`VK_AMD_shader_ballot`

## **Extension Type**

Device extension

## **Registered Extension Number**

38

## **Revision**

1

## **Extension and Version Dependencies**

- Requires Vulkan 1.0

## **Contact**

- Dominik Witczak [@dominikwitczakamd](#)

## **Last Modified Date**

2016-09-19

## **IP Status**

No known IP claims.

## **Contributors**

- Qun Lin, AMD
- Graham Sellers, AMD
- Daniel Rakos, AMD
- Rex Xu, AMD
- Dominik Witczak, AMD
- Matthäus G. Chajdas, AMD

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_AMD_shader_ballot`

## **Version History**

- Revision 1, 2016-09-19 (Dominik Witczak)
  - Initial draft

## **`VK_AMD_shader_core_properties`**

### **Name String**

`VK_AMD_shader_core_properties`

### **Extension Type**

Device extension

## Registered Extension Number

186

## Revision

2

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Contact

- Martin Dinkov [@mdinkov](#)

## Last Modified Date

2019-06-25

## IP Status

No known IP claims.

## Contributors

- Martin Dinkov, AMD
- Mattheus G. Chajdas, AMD

This extension exposes shader core properties for a target physical device through the [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#) extension. Please refer to the example below for proper usage.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_AMD`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceShaderCorePropertiesAMD](#)

## New Functions

None.

## Examples

This example retrieves the shader core properties for a physical device.

```
extern VkInstance instance;

PFN_vkGetPhysicalDeviceProperties2 pfnVkGetPhysicalDeviceProperties2 =
    reinterpret_cast<PFN_vkGetPhysicalDeviceProperties2>
        (vkGetInstanceProcAddr(instance, "vkGetPhysicalDeviceProperties2") );

VkPhysicalDeviceProperties2 general_props;
VkPhysicalDeviceShaderCorePropertiesAMD shader_core_properties;

shader_core_properties.pNext = nullptr;
shader_core_properties.sType =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_AMD;

general_props.pNext = &shader_core_properties;
general_props.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;

// After this call, shader_core_properties has been populated
pfnVkGetPhysicalDeviceProperties2(device, &general_props);

printf("Number of shader engines: %d\n",
    m_shader_core_properties.shader_engine_count =
        shader_core_properties.shaderEngineCount;
printf("Number of shader arrays: %d\n",
    m_shader_core_properties.shader_arrays_per_engine_count =
        shader_core_properties.shaderArraysPerEngineCount;
printf("Number of CUs per shader array: %d\n",
    m_shader_core_properties.compute_units_per_shader_array =
        shader_core_properties.computeUnitsPerShaderArray;
printf("Number of SIMDs per compute unit: %d\n",
    m_shader_core_propertiessimd_per_compute_unit =
        shader_core_properties.simdPerComputeUnit;
printf("Number of wavefront slots in each SIMD: %d\n",
    m_shader_core_properties.wavefronts_per_simd =
        shader_core_properties.wavefrontsPerSimd;
printf("Number of threads per wavefront: %d\n",
    m_shader_core_properties.wavefront_size =
        shader_core_properties.wavefrontSize;
printf("Number of physical SGPRs per SIMD: %d\n",
    m_shader_core_properties.sgprs_per_simd =
        shader_core_properties.sgprsPerSimd;
printf("Minimum number of SGPRs that can be allocated by a wave: %d\n",
    m_shader_core_properties.min_sgpr_allocation =
        shader_core_properties.minSgprAllocation;
printf("Number of available SGPRs: %d\n",
    m_shader_core_properties.max_sgpr_allocation =
        shader_core_properties.maxSgprAllocation;
```

```

printf("SGPRs are allocated in groups of this size: %d\n",
    m_shader_core_properties.sgpr_allocation_granularity =
    shader_core_properties.sgprAllocationGranularity;
printf("Number of physical VGPRs per SIMD: %d\n",
    m_shader_core_properties.vgprs_per_simd =
    shader_core_properties.vgprsPerSimd;
printf("Minimum number of VGPRs that can be allocated by a wave: %d\n",
    m_shader_core_properties.min_vgpr_allocation =
    shader_core_properties.minVgprAllocation;
printf("Number of available VGPRs: %d\n",
    m_shader_core_properties.max_vgpr_allocation =
    shader_core_properties.maxVgprAllocation;
printf("VGPRs are allocated in groups of this size: %d\n",
    m_shader_core_properties.vgpr_allocation_granularity =
    shader_core_properties.vgprAllocationGranularity;

```

## Version History

- Revision 2, 2019-06-25 (Matthaeus G. Chajdas)
  - Clarified the meaning of a few fields.
- Revision 1, 2018-02-15 (Martin Dinkov)
  - Initial draft.

## **VK\_AMD\_shader\_core\_properties2**

### Name String

`VK_AMD_shader_core_properties2`

### Extension Type

Device extension

### Registered Extension Number

228

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_AMD_shader_core_properties`

### Contact

- Matthaeus G. Chajdas [Panteru](#)

### Last Modified Date

2019-07-26

## IP Status

No known IP claims.

## Contributors

- Matthaeus G. Chajdas, AMD
- Tobias Hector, AMD

This extension exposes additional shader core properties for a target physical device through the [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#) extension.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_2_AMD`

## New Enums

- [VkShaderCorePropertiesFlagBitsAMD](#)

## New Structures

- [VkPhysicalDeviceShaderCoreProperties2AMD](#)

## New Functions

None.

## Examples

None.

## Version History

- Revision 1, 2019-07-26 (Matthaeus G. Chajdas)
  - Initial draft.

## **VK\_AMD\_shader\_explicit\_vertex\_parameter**

### Name String

`VK_AMD_shader_explicit_vertex_parameter`

### Extension Type

Device extension

## Registered Extension Number

22

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Qun Lin [Qlinqun](#)

## Last Modified Date

2016-05-10

## IP Status

No known IP claims.

## Contributors

- Matthaeus G. Chajdas, AMD
- Qun Lin, AMD
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Rex Xu, AMD

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_AMD_shader_explicit_vertex_parameter`

## Version History

- Revision 1, 2016-05-10 (Daniel Rakos)
  - Initial draft

## `VK_AMD_shader_fragment_mask`

### Name String

`VK_AMD_shader_fragment_mask`

### Extension Type

Device extension

## Registered Extension Number

138

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Aaron Hagan [@AaronHaganAMD](#)

## Last Modified Date

2017-08-16

## IP Status

No known IP claims.

## Dependencies

- Requires the `SPV_AMD_shader_fragment_mask` SPIR-V extension.

## Contributors

- Aaron Hagan, AMD
- Daniel Rakos, AMD
- Timothy Lottes, AMD

This extension provides efficient read access to the fragment mask in compressed multisampled color surfaces. The fragment mask is a lookup table that associates color samples with color fragment values.

From a shader, the fragment mask can be fetched with a call to `fragmentMaskFetchAMD`, which returns a single `uint` where each subsequent four bits specify the color fragment index corresponding to the color sample, starting from the least significant bit. For example, when eight color samples are used, the color fragment index for color sample 0 will be in bits 0-3 of the fragment mask, for color sample 7 the index will be in bits 28-31.

The color fragment for a particular color sample may then be fetched with the corresponding fragment mask value using the `fragmentFetchAMD` shader function.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New SPIR-V Capabilities

- `FragmentMaskAMD`

## New Structures

None.

## New Functions

None.

## Examples

This example shows a shader that queries the fragment mask from a multisampled compressed surface and uses it to query fragment values.

```
#version 450 core

#extension GL_AMD_shader_fragment_mask: enable

layout(binding = 0) uniform sampler2DMS      s2DMS;
layout(binding = 1) uniform isampler2DMSArray is2DMSArray;

layout(binding = 2, input_attachment_index = 0) uniform usubpassInputMS usubpassMS;

layout(location = 0) out vec4 fragColor;

void main()
{
    vec4 fragOne = vec4(0.0);

    uint fragMask = fragmentMaskFetchAMD(s2DMS, ivec2(2, 3));
    uint fragIndex = (fragMask & 0xF0) >> 4;
    fragOne += fragmentFetchAMD(s2DMS, ivec2(2, 3), 1);

    fragMask = fragmentMaskFetchAMD(is2DMSArray, ivec3(2, 3, 1));
    fragIndex = (fragMask & 0xF0) >> 4;
    fragOne += fragmentFetchAMD(is2DMSArray, ivec3(2, 3, 1), fragIndex);

    fragMask = fragmentMaskFetchAMD(usubpassMS);
    fragIndex = (fragMask & 0xF0) >> 4;
    fragOne += fragmentFetchAMD(usubpassMS, fragIndex);

    fragColor = fragOne;
}
```

## Version History

- Revision 1, 2017-08-16 (Aaron Hagan)
  - Initial draft

## **VK\_AMD\_shader\_image\_load\_store\_lod**

### **Name String**

`VK_AMD_shader_image_load_store_lod`

### **Extension Type**

Device extension

### **Registered Extension Number**

47

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Contact**

- Dominik Witczak [@dominikwitczakamd](#)

### **Last Modified Date**

2017-08-21

### **Interactions and External Dependencies**

- This extension requires the `SPV_AMD_shader_image_load_store_lod` SPIR-V extension.
- This extension requires `GL_AMD_shader_image_load_store_lod` for GLSL-based source languages.

### **IP Status**

No known IP claims.

### **Contributors**

- Dominik Witczak, AMD
- Qun Lin, AMD
- Rex Xu, AMD

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_AMD_shader_image_load_store_lod`

### **Version History**

- Revision 1, 2017-08-21 (Dominik Witczak)
  - Initial draft

## **VK\_AMD\_shader\_info**

**Name String**

[VK\\_AMD\\_shader\\_info](#)

**Extension Type**

Device extension

**Registered Extension Number**

43

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Jaakko Konttinen [@jaakkoamd](#)

**Last Modified Date**

2017-10-09

**IP Status**

No known IP claims.

**Contributors**

- Jaakko Konttinen, AMD

This extension adds a way to query certain information about a compiled shader which is part of a pipeline. This information may include shader disassembly, shader binary and various statistics about a shader's resource usage.

While this extension provides a mechanism for extracting this information, the details regarding the contents or format of this information are not specified by this extension and may be provided by the vendor externally.

Furthermore, all information types are optionally supported, and users should not assume every implementation supports querying every type of information.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

- [VkShaderInfoTypeAMD](#)

## New Structures

- [VkShaderResourceUsageAMD](#)
- [VkShaderStatisticsInfoAMD](#)

## New Functions

- [vkGetShaderInfoAMD](#)

## Examples

This example extracts the register usage of a fragment shader within a particular graphics pipeline:

```
extern VkDevice device;
extern VkPipeline gfxPipeline;

PFN_vkGetShaderInfoAMD pfnGetShaderInfoAMD = (PFN_vkGetShaderInfoAMD
)vkGetDeviceProcAddr(
    device, "vkGetShaderInfoAMD");

VkShaderStatisticsInfoAMD statistics = {};

size_t dataSize = sizeof(statistics);

if (pfnGetShaderInfoAMD(device,
    gfxPipeline,
    VK_SHADER_STAGE_FRAGMENT_BIT,
    VK_SHADER_INFO_TYPE_STATISTICS_AMD,
    &dataSize,
    &statistics) == VK_SUCCESS)
{
    printf("VGPR usage: %d\n", statistics.resourceUsage.numUsedVgprs);
    printf("SGPR usage: %d\n", statistics.resourceUsage.numUsedSgprs);
}
```

The following example continues the previous example by subsequently attempting to query and print shader disassembly about the fragment shader:

```

// Query disassembly size (if available)
if (pfnGetShaderInfoAMD(device,
    gfxPipeline,
    VK_SHADER_STAGE_FRAGMENT_BIT,
    VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD,
    &dataSize,
    nullptr) == VK_SUCCESS)
{
    printf("Fragment shader disassembly:\n");

    void* disassembly = malloc(dataSize);

    // Query disassembly and print
    if (pfnGetShaderInfoAMD(device,
        gfxPipeline,
        VK_SHADER_STAGE_FRAGMENT_BIT,
        VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD,
        &dataSize,
        disassembly) == VK_SUCCESS)
    {
        printf((char*)disassembly);
    }

    free(disassembly);
}

```

## Version History

- Revision 1, 2017-10-09 (Jaakko Konttinen)
  - Initial revision

## VK\_AMD\_shader\_trinary\_minmax

### Name String

`VK_AMD_shader_trinary_minmax`

### Extension Type

Device extension

### Registered Extension Number

21

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Qun Lin [Qlinqun](#)

## Last Modified Date

2016-05-10

## IP Status

No known IP claims.

## Contributors

- Matthaeus G. Chajdas, AMD
- Qun Lin, AMD
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Rex Xu, AMD

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_AMD_shader_trinary_minmax`

## Version History

- Revision 1, 2016-05-10 (Daniel Rakos)
  - Initial draft

## `VK_AMD_texture_gather_bias_lod`

### Name String

`VK_AMD_texture_gather_bias_lod`

### Extension Type

Device extension

### Registered Extension Number

42

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

## Contact

- Rex Xu [Qamdre xu](#)

## Last Modified Date

2017-03-21

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Requires the `SPV_AMD_texture_gather_bias_lod` SPIR-V extension.

## Contributors

- Dominik Witczak, AMD
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Matthaeus G. Chajdas, AMD
- Qun Lin, AMD
- Rex Xu, AMD
- Timothy Lottes, AMD

This extension adds two related features.

Firstly, support for the following SPIR-V extension in Vulkan is added:

- `SPV_AMD_texture_gather_bias_lod`

Secondly, the extension allows the application to query which formats can be used together with the new function prototypes introduced by the SPIR-V extension.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD`

## New Enums

None.

## New SPIR-V Capabilities

- `ImageGatherBiasLodAMD`

## New Structures

- `VkTextureLODGatherFormatPropertiesAMD`

## New Functions

None.

## Examples

```
struct VkTextureLODGatherFormatPropertiesAMD
{
    VkStructureType sType;
    const void*     pNext;
    VkBool32        supportsTextureGatherLODBiasAMD;
};

// -----
-- 
// How to detect if an image format can be used with the new function prototypes.
VkPhysicalDeviceImageFormatInfo2    formatInfo;
VkImageFormatProperties2            formatProps;
VkTextureLODGatherFormatPropertiesAMD textureLODGatherSupport;

textureLODGatherSupport.sType =
VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD;
textureLODGatherSupport.pNext = nullptr;

formatInfo.sType    = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2;
formatInfo.pNext    = nullptr;
formatInfo.format   = ...;
formatInfo.type     = ...;
formatInfo.tiling   = ...;
formatInfo.usage    = ...;
formatInfo.flags    = ...;

formatProps.sType = VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2;
formatProps.pNext = &textureLODGatherSupport;

vkGetPhysicalDeviceImageFormatProperties2(physical_device, &formatInfo, &formatProps);

if (textureLODGatherSupport.supportsTextureGatherLODBiasAMD == VK_TRUE)
{
    // physical device supports SPV_AMD_texture_gather_bias_lod for the specified
    // format configuration.
}
else
{
    // physical device does not support SPV_AMD_texture_gather_bias_lod for the
    // specified format configuration.
}
```

## Version History

- Revision 1, 2017-03-21 (Dominik Witczak)
  - Initial draft

## **VK\_ANDROID\_external\_memory\_android\_hardware\_buffer**

### Name String

`VK_ANDROID_external_memory_android_hardware_buffer`

### Extension Type

Device extension

### Registered Extension Number

130

### Revision

3

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_sampler_ycbcr_conversion`
- Requires `VK_KHR_external_memory`
- Requires `VK_EXT_queue_family_foreign`

### Contact

- Jesse Hall 

### Last Modified Date

2019-08-27

### IP Status

No known IP claims.

### Contributors

- Ray Smith, ARM
- Chad Versace, Google
- Jesse Hall, Google
- Tobias Hector, Imagination
- James Jones, NVIDIA
- Tony Zlatinski, NVIDIA
- Matthew Netsch, Qualcomm
- Andrew Garrard, Samsung

This extension enables an application to import Android [AHardwareBuffer](#) objects created outside of the Vulkan device into Vulkan memory objects, where they **can** be bound to images and buffers. It also allows exporting an [AHardwareBuffer](#) from a Vulkan memory object for symmetry with other operating systems. But since not all [AHardwareBuffer](#) usages and formats have Vulkan equivalents, exporting from Vulkan provides strictly less functionality than creating the [AHardwareBuffer](#) externally and importing it.

Some [AHardwareBuffer](#) images have implementation-defined *external formats* that **may** not correspond to Vulkan formats. Sampler Y'CbCr conversion **can** be used to sample from these images and convert them to a known color space.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_USAGE_ANDROID`
  - `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_PROPERTIES_ANDROID`
  - `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_FORMAT_PROPERTIES_ANDROID`
  - `VK_STRUCTURE_TYPE_IMPORT_ANDROID_HARDWARE_BUFFER_INFO_ANDROID`
  - `VK_STRUCTURE_TYPE_MEMORY_GET_ANDROID_HARDWARE_BUFFER_INFO_ANDROID`
  - `VK_STRUCTURE_TYPE_EXTERNAL_FORMAT_ANDROID`
- Extending [VkExternalMemoryHandleTypeFlagBits](#):
  - `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`

## New Enums

None.

## New Structs

- [VkAndroidHardwareBufferUsageANDROID](#)
- [VkAndroidHardwareBufferPropertiesANDROID](#)
- [VkAndroidHardwareBufferFormatPropertiesANDROID](#)
- [VkImportAndroidHardwareBufferInfoANDROID](#)
- [VkMemoryGetAndroidHardwareBufferInfoANDROID](#)
- [VkExternalFormatANDROID](#)

## New Functions

- [vkGetAndroidHardwareBufferPropertiesANDROID](#)
- [vkGetMemoryAndroidHardwareBufferANDROID](#)

## Issues

1) Other external memory objects are represented as weakly-typed handles (e.g. Win32 `HANDLE` or POSIX file descriptor), and require a handle type parameter along with handles. `AHardwareBuffer` is strongly typed, so naming the handle type is redundant. Does symmetry justify adding handle type parameters/fields anyway?

**RESOLVED:** No. The handle type is already provided in places that treat external memory objects generically. In the places we would add it, the application code that would have to provide the handle type value is already dealing with `AHardwareBuffer`-specific commands/structures; the extra symmetry would not be enough to make that code generic.

2) The internal layout and therefore size of a `AHardwareBuffer` image may depend on native usage flags that do not have corresponding Vulkan counterparts. Do we provide this info to `vkCreateImage` somehow, or allow the allocation size reported by `vkGetImageMemoryRequirements` to be approximate?

**RESOLVED:** Allow the allocation size to be unspecified when allocating the memory. It has to work this way for exported image memory anyway, since `AHardwareBuffer` allocation happens in `vkAllocateMemory`, and internally is performed by a separate HAL, not the Vulkan implementation itself. There is a similar issue with `vkGetImageSubresourceLayout`: the layout is determined by the allocator HAL, so it is not known until the image is bound to memory.

3) Should the result of sampling an external-format image with the suggested  $Y' C_b C_r$  conversion parameters yield the same results as using a `samplerExternalOES` in OpenGL ES?

**RESOLVED:** This would be desirable, so that apps converting from OpenGL ES to Vulkan could get the same output given the same input. But since sampling and conversion from  $Y' C_b C_r$  images is so loosely defined in OpenGL ES, multiple implementations do it in a way that doesn't conform to Vulkan's requirements. Modifying the OpenGL ES implementation would be difficult, and would change the output of existing unmodified applications. Changing the output only for applications that are being modified gives developers the chance to notice and mitigate any problems. Implementations are encouraged to minimize differences as much as possible without causing compatibility problems for existing OpenGL ES applications or violating Vulkan requirements.

4) Should an `AHardwareBuffer` with `AHARDWAREBUFFER_USAGE_CPU_*` usage be mappable in Vulkan? Should it be possible to export an `AHardwareBuffers` with such usage?

**RESOLVED:** Optional, and mapping in Vulkan is not the same as `AHardwareBuffer_lock`. The semantics of these are different: mapping in memory is persistent, just gives a raw view of the memory contents, and does not involve ownership. `AHardwareBuffer_lock` gives the host exclusive access to the buffer, is temporary, and allows for reformatting copy-in/copy-out. Implementations are not required to support host-visible memory types for imported Android hardware buffers or resources backed by them. If a host-visible memory type is supported and used, the memory can be mapped in Vulkan, but doing so follows Vulkan semantics: it is just a raw view of the data and does not imply ownership (this means implementations must not internally call `AHardwareBuffer_lock` to implement `vkMapMemory`, or assume the application has done so). Implementations are not required to support linear-tiled images backed by Android hardware buffers, even if the `AHardwareBuffer` has CPU usage. There is no reliable way to allocate memory in Vulkan that can be

exported to a [AHardwareBuffer](#) with CPU usage.

5) Android may add new [AHardwareBuffer](#) formats and usage flags over time. Can reference to them be added to this extension, or do they need a new extension?

RESOLVED: This extension can document the interaction between the new AHB formats/usages and existing Vulkan features. No new Vulkan features or implementation requirements can be added. The extension version number will be incremented when this additional documentation is added, but the version number does not indicate that an implementaiton supports Vulkan memory or resources that map to the new [AHardwareBuffer](#) features: support for that must be queried with `vkGetPhysicalDeviceImageFormatProperties2` or is implied by successfully allocating a [AHardwareBuffer](#) outside of Vulkan that uses the new feature and has a GPU usage flag.

In essence, these are new features added to a new Android API level, rather than new Vulkan features. The extension will only document how existing Vulkan features map to that new Android feature.

## Version History

- Revision 3, 2019-08-27 (Jon Leech)
  - Update revision history to correspond to XML version number
- Revision 2, 2018-04-09 (Petr Kraus)
  - Markup fixes and remove incorrect Draft status
- Revision 1, 2018-03-04 (Jesse Hall)
  - Initial version

## VK\_FUCHSIA\_imagepipe\_surface

### Name String

`VK_FUCHSIA_imagepipe_surface`

### Extension Type

Instance extension

### Registered Extension Number

215

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### Contact

- Craig Stout [craig.stout@fuchsia.com](mailto:craig.stout@fuchsia.com)

## Last Modified Date

2018-07-27

## IP Status

No known IP claims.

## Contributors

- Craig Stout, Google
- Ian Elliott, Google
- Jesse Hall, Google

The `VK_FUCHSIA_imagepipe_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to a Fuchsia `imagePipeHandle`.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_IMAGEPIPE_SURFACE_CREATE_INFO_FUCHSIA`

## New Enums

None

## New Structures

- `VkImagePipeSurfaceCreateInfoFUCHSIA`

## New Functions

- `vkCreateImagePipeSurfaceFUCHSIA`

## Issues

None

## Version History

- Revision 1, 2018-07-27 (Craig Stout)
  - Initial draft.

## **VK\_GGP\_frame\_token**

### **Name String**

`VK_GGP_frame_token`

### **Extension Type**

Device extension

### **Registered Extension Number**

192

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_swapchain`
- Requires `VK_GGP_stream_descriptor_surface`

### **Contact**

- Jean-Francois Roy [Ojfroy](#)

### **Last Modified Date**

2019-01-28

### **IP Status**

No known IP claims.

### **Contributors**

- Jean-Francois Roy, Google
- Richard O'Grady, Google

This extension allows an application that uses the `VK_KHR_swapchain` extension in combination with a Google Games Platform surface provided by the `VK_GGP_stream_descriptor_surface` extension to associate a Google Games Platform frame token with a present operation.

## **New Object Types**

None.

## **New Enum Constants**

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PRESENT_FRAME_TOKEN_GGP`

## New Enums

None.

## New Structures

- [VkPresentFrameTokenGGP](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2018-11-26 (Jean-Francois Roy)
  - Initial revision.

## VK\_GGP\_stream\_descriptor\_surface

### Name String

`VK_GGP_stream_descriptor_surface`

### Extension Type

Instance extension

### Registered Extension Number

50

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### Contact

- Jean-Francois Roy [Qjfroy](#)

### Last Modified Date

2019-01-28

### IP Status

No known IP claims.

## Contributors

- Jean-Francois Roy, Google
- Brad Grantham, Google
- Connor Smith, Google
- Cort Stratton, Google
- Hai Nguyen, Google
- Ian Elliott, Google
- Jesse Hall, Google
- Jim Ray, Google
- Katherine Wu, Google
- Kaye Mason, Google
- Kuangye Guo, Google
- Mark Segal, Google
- Nicholas Vining, Google
- Paul Lalonde, Google
- Richard O'Grady, Google

The `VK_GGP_STREAM_DESCRIPTOR_SURFACE` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_SURFACE` extension) that refers to a Google Games Platform `GgpStreamDescriptor`.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_STREAM_DESCRIPTOR_SURFACE_CREATE_INFO_GGP`

## New Enums

None.

## New Structures

- `VkStreamDescriptorSurfaceCreateInfoGGP`

## New Functions

- `vkCreateStreamDescriptorSurfaceGGP`

## Issues

None.

## Version History

- Revision 1, 2018-11-26 (Jean-Francois Roy)
  - Initial revision.

## VK\_GOOGLE\_decorate\_string

### Name String

`VK_GOOGLE_decorate_string`

### Extension Type

Device extension

### Registered Extension Number

225

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Hai Nguyen [@chaoticbob](#)

### Last Modified Date

2018-07-09

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Requires the `SPV_GOOGLE_decorate_string` SPIR-V extension.

### Contributors

- Hai Nguyen, Google
- Neil Henning, AMD

The `VK_GOOGLE_decorate_string` extension allows use of the `SPV_GOOGLE_decorate_string` extension in SPIR-V shader modules.

## New Enum Constants

None.

## New Structures

None.

## New SPIR-V Capabilities

None.

## Issues

### Version History

- Revision 1, 2018-07-09 (Neil Henning)
  - Initial draft

## VK\_GOOGLE\_display\_timing

### Name String

`VK_GOOGLE_display_timing`

### Extension Type

Device extension

### Registered Extension Number

93

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_swapchain](#)

### Contact

- Ian Elliott [@ianelliottus](#)

### Last Modified Date

2017-02-14

### IP Status

No known IP claims.

### Contributors

- Ian Elliott, Google
- Jesse Hall, Google

This device extension allows an application that uses the [VK\\_KHR\\_swapchain](#) extension to obtain information about the presentation engine's display, to obtain timing information about each

present, and to schedule a present to happen no earlier than a desired time. An application can use this to minimize various visual anomalies (e.g. stuttering).

Traditional game and real-time animation applications need to correctly position their geometry for when the presentable image will be presented to the user. To accomplish this, applications need various timing information about the presentation engine's display. They need to know when presentable images were actually presented, and when they could have been presented. Applications also need to tell the presentation engine to display an image no sooner than a given time. This allows the application to avoid stuttering, so the animation looks smooth to the user.

This extension treats variable-refresh-rate (VRR) displays as if they are fixed-refresh-rate (FRR) displays.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PRESENT_TIMES_INFO_GOOGLE`

## New Enums

None.

## New Structures

- [VkRefreshCycleDurationGOOGLE](#)
- [VkPastPresentationTimingGOOGLE](#)
- [VkPresentTimesInfoGOOGLE](#)
- [VkPresentTimeGOOGLE](#)

## New Functions

- [vkGetRefreshCycleDurationGOOGLE](#)
- [vkGetPastPresentationTimingGOOGLE](#)

## Issues

None.

## Examples

### Note



The example code for this extension (like the `VK_KHR_surface` and `VK_GOOGLE_display_timing` extensions) is contained in the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: <https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c> ).

## Version History

- Revision 1, 2017-02-14 (Ian Elliott)
  - Internal revisions

## `VK_GOOGLE_hlsl_functionality1`

### Name String

`VK_GOOGLE_hlsl_functionality1`

### Extension Type

Device extension

### Registered Extension Number

224

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Hai Nguyen [@chaoticbob](#)

### Last Modified Date

2018-07-09

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Requires the `SPV_GOOGLE_hlsl_functionality1` SPIR-V extension.

### Contributors

- Hai Nguyen, Google
- Neil Henning, AMD

The `VK_GOOGLE_hlsl_functionality1` extension allows use of the `SPV_GOOGLE_hlsl_functionality1` extension in SPIR-V shader modules.

## New Enum Constants

None.

## New Structures

None.

## New SPIR-V Capabilities

None.

## Issues

### Version History

- Revision 1, 2018-07-09 (Neil Henning)
  - Initial draft

## `VK_GOOGLE_user_type`

### Name String

`VK_GOOGLE_user_type`

### Extension Type

Device extension

### Registered Extension Number

290

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Kaye Mason [Ochaleur](#)

### Last Modified Date

2019-07-09

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Requires the `SPV_GOOGLE_user_type` SPIR-V extension.

## Contributors

- Kaye Mason, Google
- Hai Nguyen, Google

The `VK_GOOGLE_user_type` extension allows use of the `SPV_GOOGLE_user_type` extension in SPIR-V shader modules.

## New Enum Constants

None.

## New Structures

None.

## New SPIR-V Capabilities

None.

## Issues

### Version History

- Revision 1, 2019-09-07 (Kaye Mason)
  - Initial draft

## `VK_IMG_filter_cubic`

### Name String

`VK_IMG_filter_cubic`

### Extension Type

Device extension

### Registered Extension Number

16

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Tobias Hector [@tobbski](#)

### Last Modified Date

2016-02-23

## Contributors

- Tobias Hector, Imagination Technologies

`VK_IMG_filter_cubic` adds an additional, high quality cubic filtering mode to Vulkan, using a Catmull-Rom bicubic filter. Performing this kind of filtering can be done in a shader by using 16 samples and a number of instructions, but this can be inefficient. The cubic filter mode exposes an optimized high quality texture sampling using fixed texture sampling functionality.

## New Enum Constants

- Extending `VkFilter`:
  - `VK_FILTER_CUBIC_IMG`
- Extending `VkFormatFeatureFlagBits`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG`

## Example

Creating a sampler with the new filter for both magnification and minification

```
VkSamplerCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO // sType
    // Other members set to application-desired values
};

createInfo.magFilter = VK_FILTER_CUBIC_IMG;
createInfo.minFilter = VK_FILTER_CUBIC_IMG;

VkSampler sampler;
VkResult result = vkCreateSampler(
    device,
    &createInfo,
    &sampler);
```

## Version History

- Revision 1, 2016-02-23 (Tobias Hector)
  - Initial version

## VK\_IMG\_format\_pvrtc

### Name String

`VK_IMG_format_pvrtc`

### Extension Type

Device extension

## Registered Extension Number

55

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Stuart Smith

## Last Modified Date

2019-09-02

## IP Status

Imagination Technologies Proprietary

## Contributors

- Stuart Smith, Imagination Technologies

`VK_IMG_format_pvrtc` provides additional texture compression functionality specific to Imagination Technologies PowerVR Texture compression format (called PVRTC).

## New Object Types

None.

## New Enum Constants

- Extending `VkFormat`:
  - `VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG`
  - `VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG`
  - `VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG`
  - `VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG`
  - `VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG`
  - `VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG`
  - `VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG`
  - `VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG`

## New Enum Constants

None.

## New Structures

None.

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2019-09-02 (Stuart Smith)
  - Initial version

## VK\_INTEL\_performance\_query

### Name String

`VK_INTEL_performance_query`

### Extension Type

Device extension

### Registered Extension Number

211

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Lionel Landwerlin [@landwerlin](#)

### Last Modified Date

2018-05-16

### IP Status

No known IP claims.

### Contributors

- Lionel Landwerlin, Intel
- Piotr Maciejewski, Intel

This extension allows an application to capture performance data to be interpreted by a external application or library.

Such a library is available at : <https://github.com/intel/metrics-discovery>

Performance analysis tools such as GPA (<https://software.intel.com/en-us/gpa>) make use of this

extension and the metrics-discovery library to present the data in a human readable way.

## New Object Types

- [VkPerformanceConfigurationINTEL](#)

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO_INTEL`
  - `VK_STRUCTURE_TYPE_INITIALIZE_PERFORMANCE_API_INFO_INTEL`
  - `VK_STRUCTURE_TYPE_PERFORMANCE_MARKER_INFO_INTEL`
  - `VK_STRUCTURE_TYPE_PERFORMANCE_STREAM_MARKER_INFO_INTEL`
  - `VK_STRUCTURE_TYPE_PERFORMANCE_OVERRIDE_INFO_INTEL`
  - `VK_STRUCTURE_TYPE_PERFORMANCE_CONFIGURATION_ACQUIRE_INFO_INTEL`
- Extending [VkQueryType](#):
  - `VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL`

## New Enums

- [VkPerformanceConfigurationTypeINTEL](#)
- [VkQueryPoolSamplingModeINTEL](#)
- [VkPerformanceOverrideTypeINTEL](#)
- [VkPerformanceParameterTypeINTEL](#)
- [VkPerformanceValueTypeINTEL](#)

## New Structures

- [VkPerformanceValueINTEL](#)
- [VkInitializePerformanceApiInfoINTEL](#)
- [VkQueryPoolCreateInfoINTEL](#)
- [VkPerformanceMarkerCreateInfoINTEL](#)
- [VkPerformanceStreamMarkerCreateInfoINTEL](#)
- [VkPerformanceOverrideCreateInfoINTEL](#)
- [VkPerformanceConfigurationAcquireCreateInfoINTEL](#)

## New Functions

- [vkInitializePerformanceApiINTEL](#)
- [vkUninitializePerformanceApiINTEL](#)
- [vkCmdSetPerformanceMarkerINTEL](#)
- [vkCmdSetPerformanceOverrideINTEL](#)

- [vkCmdSetPerformanceStreamMarkerINTEL](#)
- [vkAcquirePerformanceConfigurationINTEL](#)
- [vkReleasePerformanceConfigurationINTEL](#)
- [vkQueueSetPerformanceConfigurationINTEL](#)
- [vkGetPerformanceParameterINTEL](#)

## Issues

None.

## Example Code

```
// A previously created device
VkDevice device;

// A queue from from device
VkQueue queue;

VkInitializePerformanceApiInfoINTEL performanceApiInfoIntel = {
    VK_STRUCTURE_TYPE_INITIALIZE_PERFORMANCE_API_INFO_INTEL,
    NULL,
    NULL
};

vkInitializePerformanceApiINTEL(
    device,
    &performanceApiInfoIntel);

VkQueryPoolCreateInfoINTEL queryPoolIntel = {
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO_INTEL,
    NULL,
    VK_QUERY_POOL_SAMPLING_MODE_MANUAL_INTEL,
};

VkQueryPoolCreateInfo queryPoolCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO,
    &queryPoolIntel,
    0,
    VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL,
    1,
    0
};

VkQueryPool queryPool;

VkResult result = vkCreateQueryPool(
    device,
    &queryPoolCreateInfo,
```

```

NULL,
&queryPool);

assert(VK_SUCCESS == result);

// A command buffer we want to record counters on
VkCommandBuffer commandBuffer;

VkCommandBufferBeginInfo commandBufferBeginInfo = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    NULL,
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT,
    NULL
};

result = vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);

assert(VK_SUCCESS == result);

vkCmdResetQueryPool(
    commandBuffer,
    queryPool,
    0,
    1);

vkCmdBeginQuery(
    commandBuffer,
    queryPool,
    0,
    0);

// Perform the commands you want to get performance information on
// ...

// Perform a barrier to ensure all previous commands were complete before
// ending the query
vkCmdPipelineBarrier(commandBuffer,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    0,
    0,
    NULL,
    0,
    NULL,
    0,
    NULL);

vkCmdEndQuery(
    commandBuffer,
    queryPool,
    0);

```

```

result = vkEndCommandBuffer(commandBuffer);

assert(VK_SUCCESS == result);

VkPerformanceConfigurationAcquireInfoINTEL performanceConfigurationAcquireInfo = {
    VK_STRUCTURE_TYPE_PERFORMANCE_CONFIGURATION_ACQUIRE_INFO_INTEL,
    NULL,
    VK_PERFORMANCE_CONFIGURATION_TYPE_COMMAND_QUEUE_METRICS_DISCOVERY_ACTIVATED_INTEL
};

VkPerformanceConfigurationINTEL performanceConfigurationIntel;

result = vkAcquirePerformanceConfigurationINTEL(
    device,
    &performanceConfigurationAcquireInfo,
    &performanceConfigurationIntel);

vkQueueSetPerformanceConfigurationINTEL(queue, performanceConfigurationIntel);

assert(VK_SUCCESS == result);

// Submit the command buffer and wait for its completion
// ...

result = vkReleasePerformanceConfigurationINTEL(
    device,
    performanceConfigurationIntel);

assert(VK_SUCCESS == result);

// Get the report size from metrics-discovery's QueryReportSize

result = vkGetQueryPoolResults(
    device,
    queryPool,
    0, 1, QueryReportSize,
    data, QueryReportSize, 0);

assert(VK_SUCCESS == result);

// The data can then be passed back to metrics-discovery from which
// human readable values can be queried.

```

## Version History

- Revision 1, 2018-05-16 (Lionel Landwerlin)
  - Initial revision

## **VK\_INTEL\_shader\_integer\_functions2**

### **Name String**

`VK_INTEL_shader_integer_functions2`

### **Extension Type**

Device extension

### **Registered Extension Number**

210

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### **Contact**

- Ian Romanick [@ianromanick](#)

### **Last Modified Date**

2019-04-30

### **IP Status**

No known IP claims.

### **Contributors**

- Ian Romanick, Intel
- Ben Ashbaugh, Intel

This extension adds support for several new integer instructions in SPIR-V for use in graphics shaders. Many of these instructions have pre-existing counterparts in the Kernel environment.

The added integer functions are defined by the `SPV_INTEL_shader_integer_functions` SPIR-V extension and can be used with the `GL_INTEL_shader_integer_functions2` GLSL extension.

## **New Object Types**

None.

## **New Enum Constants**

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_FUNCTIONS_2_FEATURES_INTEL`

## New Enums

None.

## New Structures

- Extending `VkPhysicalDeviceFeatures2`:
  - `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL`

## New Functions

None.

## New SPIR-V Capabilities

- `ShaderIntegerFunctions2INTEL`

## Issues

None.

## Version History

- Revision 1, 2019-04-30 (Ian Romanick)
  - Initial draft

## VK\_MVK\_ios\_surface

### Name String

`VK_MVK_ios_surface`

### Extension Type

Instance extension

### Registered Extension Number

123

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

### Contact

- Bill Hollings [@billhollings](#)

## Last Modified Date

2017-02-24

## IP Status

No known IP claims.

## Contributors

- Bill Hollings, The Brenwill Workshop Ltd.

The `VK_MVK_ios_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to a `UIView`, the native surface type of iOS, which is underpinned by a `CAMetalLayer`, to support rendering to the surface using Apple's Metal framework.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_IOS_SURFACE_CREATE_INFO_MVK`

## New Enums

None.

## New Structures

- `VkIOSSurfaceCreateInfoMVK`

## New Functions

- `vkCreateIOSSurfaceMVK`

## Issues

None.

## Version History

- Revision 1, 2017-02-15 (Bill Hollings)
  - Initial draft.
- Revision 2, 2017-02-24 (Bill Hollings)
  - Minor syntax fix to emphasize firm requirement for `UIView` to be backed by a `CAMetalLayer`.

## **VK\_MVK\_macos\_surface**

### **Name String**

`VK_MVK_macos_surface`

### **Extension Type**

Instance extension

### **Registered Extension Number**

124

### **Revision**

2

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### **Contact**

- Bill Hollings [billhollings](#)

### **Last Modified Date**

2017-02-24

### **IP Status**

No known IP claims.

### **Contributors**

- Bill Hollings, The Brenwill Workshop Ltd.

The `VK_MVK_macos_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the [VK\\_KHR\\_surface](#) extension) that refers to an `NSView`, the native surface type of macOS, which is underpinned by a `CAMetalLayer`, to support rendering to the surface using Apple's Metal framework.

## **New Object Types**

None.

## **New Enum Constants**

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK`

## **New Enums**

None.

## New Structures

- [VkMacOSSurfaceCreateInfoMVK](#)

## New Functions

- [vkCreateMacOSSurfaceMVK](#)

## Issues

None.

## Version History

- Revision 1, 2017-02-15 (Bill Hollings)
  - Initial draft.
- Revision 2, 2017-02-24 (Bill Hollings)
  - Minor syntax fix to emphasize firm requirement for NSView to be backed by a CAMetalLayer.

## VK\_NN\_vi\_surface

### Name String

`VK_NN_vi_surface`

### Extension Type

Instance extension

### Registered Extension Number

63

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_surface](#)

### Contact

- Mathias Heyer mheyer

### Last Modified Date

2016-12-02

### IP Status

No known IP claims.

## Contributors

- Mathias Heyer, NVIDIA
- Michael Chock, NVIDIA
- Yasuhiro Yoshioka, Nintendo
- Daniel Koch, NVIDIA

The `VK_NN_vi_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) associated with an `nn::vi::Layer`.

## New Object Types

None

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_VI_SURFACE_CREATE_INFO_NN`

## New Enums

None

## New Structures

- `VkViSurfaceCreateInfoNN`

## New Functions

- `vkCreateViSurfaceNN`

## Issues

1) Does VI need a way to query for compatibility between a particular physical device (and queue family?) and a specific VI display?

**RESOLVED:** No. It is currently always assumed that the device and display will always be compatible.

2) `VkViSurfaceCreateInfoNN::pWindow` is intended to store an `nn::vi::NativeWindowHandle`, but its declared type is a bare `void*` to store the window handle. Why the discrepancy?

**RESOLVED:** It is for C compatibility. The definition for the VI native window handle type is defined inside the `nn::vi` C++ namespace. This prevents its use in C source files. `nn::vi::NativeWindowHandle` is always defined to be `void*`, so this extension uses `void*` to match.

## Version History

- Revision 1, 2016-12-2 (Michael Chock)

- Initial draft.

## **VK\_NVX\_device\_generated\_commands**

### **Name String**

`VK_NVX_device_generated_commands`

### **Extension Type**

Device extension

### **Registered Extension Number**

87

### **Revision**

3

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Contact**

- Christoph Kubisch [@pixeljetstream](#)

### **Last Modified Date**

2017-07-25

### **Contributors**

- Pierre Boudier, NVIDIA
- Christoph Kubisch, NVIDIA
- Mathias Schott, NVIDIA
- Jeff Bolz, NVIDIA
- Eric Werness, NVIDIA
- Detlef Roettger, NVIDIA
- Daniel Koch, NVIDIA
- Chris Hebert, NVIDIA

This extension allows the device to generate a number of critical commands for command buffers.

When rendering a large number of objects, the device can be leveraged to implement a number of critical functions, like updating matrices, or implementing occlusion culling, frustum culling, front to back sorting, etc. Implementing those on the device does not require any special extension, since an application is free to define its own data structure, and just process them using shaders.

However, if the application desires to quickly kick off the rendering of the final stream of objects, then unextended Vulkan forces the application to read back the processed stream and issue graphics command from the host. For very large scenes, the synchronization overhead, and cost to generate the command buffer can become the bottleneck. This extension allows an application to

generate a device side stream of state changes and commands, and convert it efficiently into a command buffer without having to read it back on the host.

Furthermore, it allows incremental changes to such command buffers by manipulating only partial sections of a command stream—for example pipeline bindings. Unextended Vulkan requires re-creation of entire command buffers in such scenario, or updates synchronized on the host.

The intended usage for this extension is for the application to:

- create its objects as in unextended Vulkan
- create a [VkObjectTableNVX](#), and register the various Vulkan objects that are needed to evaluate the input parameters.
- create a [VkIndirectCommandsLayoutNVX](#), which lists the [VkIndirectCommandsTokenTypeNVX](#) it wants to dynamically change as atomic command sequence. This step likely involves some internal device code compilation, since the intent is for the GPU to generate the command buffer in the pipeline.
- fill the input buffers with the data for each of the inputs it needs. Each input is an array that will be filled with an index in the object table, instead of using CPU pointers.
- set up a target secondary command buffer
- reserve command buffer space via [vkCmdReserveSpaceForCommandsNVX](#) in a target command buffer at the position you want the generated commands to be executed.
- call [vkCmdProcessCommandsNVX](#) to create the actual device commands for all sequences based on the array contents into a provided target command buffer.
- execute the target command buffer like a regular secondary command buffer

For each draw/dispatch, the following can be specified:

- a different pipeline state object
- a number of descriptor sets, with dynamic offsets
- a number of vertex buffer bindings, with an optional dynamic offset
- a different index buffer, with an optional dynamic offset

Applications **should** register a small number of objects, and use dynamic offsets whenever possible.

While the GPU can be faster than a CPU to generate the commands, it may not happen asynchronously, therefore the primary use-case is generating “less” total work (occlusion culling, classification to use specialized shaders, etc.).

## New Object Types

- [VkObjectTableNVX](#)
- [VkIndirectCommandsLayoutNVX](#)

## New Flag Types

- [VkIndirectCommandsLayoutUsageFlagsNVX](#)
- [VkObjectEntryUsageFlagsNVX](#)

## New Enum Constants

Extending [VkStructureType](#):

- `VK_STRUCTURE_TYPE_OBJECT_TABLE_CREATE_INFO_NVX`
- `VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_CREATE_INFO_NVX`
- `VK_STRUCTURE_TYPE_CMD_PROCESS_COMMANDS_INFO_NVX`
- `VK_STRUCTURE_TYPE_CMD_RESERVE_SPACE_FOR_COMMANDS_INFO_NVX`
- `VK_STRUCTURE_TYPE_DEVICE_GENERATED_COMMANDS_LIMITS_NVX`
- `VK_STRUCTURE_TYPE_DEVICE_GENERATED_COMMANDS_FEATURES_NVX`

Extending [VkPipelineStageFlagBits](#):

- `VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX`

Extending [VkAccessFlagBits](#):

- `VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX`
- `VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX`

## New Enums

- [VkIndirectCommandsLayoutUsageFlagBitsNVX](#)
- [VkIndirectCommandsTokenTypeNVX](#)
- [VkObjectEntryUsageFlagBitsNVX](#)
- [VkObjectEntryTypeNVX](#)

## New Structures

- [VkDeviceGeneratedCommandsFeaturesNVX](#)
- [VkDeviceGeneratedCommandsLimitsNVX](#)
- [VkIndirectCommandsTokenNVX](#)
- [VkIndirectCommandsLayoutTokenNVX](#)
- [VkIndirectCommandsLayoutCreateInfoNVX](#)
- [VkCmdProcessCommandsInfoNVX](#)
- [VkCmdReserveSpaceForCommandsInfoNVX](#)
- [VkObjectTableCreateInfoNVX](#)
- [VkObjectTableEntryNVX](#)
- [VkObjectTablePipelineEntryNVX](#)

- [VkObjectTableDescriptorSetEntryNVX](#)
- [VkObjectTableVertexBufferEntryNVX](#)
- [VkObjectTableIndexBufferEntryNVX](#)
- [VkObjectTablePushConstantEntryNVX](#)

## New Functions

- [vkCmdProcessCommandsNVX](#)
- [vkCmdReserveSpaceForCommandsNVX](#)
- [vkCreateIndirectCommandsLayoutNVX](#)
- [vkDestroyIndirectCommandsLayoutNVX](#)
- [vkCreateObjectTableNVX](#)
- [vkDestroyObjectTableNVX](#)
- [vkRegisterObjectsNVX](#)
- [vkUnregisterObjectsNVX](#)
- [vkGetPhysicalDeviceGeneratedCommandsPropertiesNVX](#)

## Issues

1) How to name this extension ?

**RESOLVED:** `VK_NVX_device_generated_commands`

As usual, one of the hardest issues ;)

Alternatives: `VK_gpu_commands`, `VK_execute_commands`, `VK_device_commands`, `VK_device_execute_commands`,  
`VK_device_execute`,                    `VK_device_created_commands`,                    `VK_device_recorded_commands`,  
`VK_device_generated_commands`

2) Should we use serial tokens or redundant sequence description?

Similarly to [VkPipeline](#), signatures have the most likelihood to be cross-vendor adoptable. They also benefit from being processable in parallel.

3) How to name sequence description

`ExecuteCommandSignature` is a bit long. Maybe just `ExecuteSignature`, or actually more following Vulkan nomenclature: [VkIndirectCommandsLayoutNVX](#).

4) Do we want to provide `indirectCommands` inputs with layout or at `indirectCommands` time?

Separate layout from data as Vulkan does. Provide full flexibility for `indirectCommands`.

5) Should the input be provided as SoA or AoS?

It is desirable for the application to reuse the list of objects and render them with some kind of an override. This can be done by just selecting a different input for a push constant or a descriptor set,

if they are defined as independent arrays. If the data was interleaved, this would not be as easily possible.

Allowing input divisors can also reduce the conservative command buffer allocation.

6) How do we know the size of the GPU command buffer generated by [vkCmdProcessCommandsNVX](#)?

`maxSequenceCount` can give an upper estimate, even if the actual count is sourced from the gpu buffer at (buffer, countOffset). As such `maxSequenceCount` must always be set correctly.

Developers are encouraged to make well use the [VkIndirectCommandsLayoutNVX](#)'s `pTokens[]].divisor`, as they allow less conservative storage costs. Especially pipeline changes on a per-draw basis can be costly memory wise.

7) How to deal with dynamic offsets in DescriptorSets?

Maybe additional token `VK_EXECUTE_DESCRIPTOR_SET_OFFSET_COMMAND_NVX` that works for a “single dynamic buffer” descriptor set and then use (32 bit tableEntry + 32bit offset)

added dynamicCount field, variable sized input

8) Should we allow updates to the object table, similar to DescriptorSet?

Desired yes, people may change “material” shaders and not want to recreate the entire register table. However the developer must ensure to not overwrite a registered objectIndex while it is still being used.

9) Should we allow dynamic state changes?

Seems a bit excessive for “per-draw” type of scenario, but GPU could partition work itself with viewport/scissor...

10) How do we allow re-using already “filled” `indirectCommands` buffers?

just use a [VkCommandBuffer](#) for the output, and it can be reused easily.

11) How portable should such re-use be?

Same as secondary command buffer

12) Should `sequenceOrdered` be part of [IndirectCommandsLayout](#) or [vkCmdProcessCommandsNVX](#)?

Seems better for [IndirectCommandsLayout](#), as that is when most heavy lifting in terms of internal device code generation is done.

13) Under which conditions is [vkCmdProcessCommandsNVX](#) legal?

Options:

a) on the host command buffer like a regular draw call

- b) `vkCmdProcessCommandsNVX` makes use `VkCommandBufferBeginInfo` and serves as `vkBeginCommandBuffer` / `vkEndCommandBuffer` implicitly.
- c) The `targetCommandbuffer` must be inside the “begin” state already at the moment of being passed. This very likely suggests a new `VkCommandBufferUsageFlags` `VK_COMMAND_BUFFER_USAGE_DEVICE_GENERATED_BIT`.

d) The `targetCommandbuffer` must reserve space via a new function.

used a) and d).

14) What if different pipelines have different DescriptorSetLayouts at a certain set unit that mismatches in `token.dynamicCount`?

Considered legal, as long as the maximum dynamic count of all used DescriptorSetLayouts is provided.

15) Should we add “strides” to input arrays, so that “Array of Structures” type setups can be supported more easily?

Maybe provide a usage flag for packed tokens stream (all inputs from same buffer, implicit stride).

No, given performance test was worse.

16) Should we allow re-using the target command buffer directly, without need to reset command buffer?

YES: new api `vkCmdReserveSpaceForCommandsNVX`.

17) Is `vkCmdProcessCommandsNVX` copying the input data or referencing it ?

There are multiple implementations possible:

- one could have some emulation code that parse the inputs, and generates an output command buffer, therefore copying the inputs.
- one could just reference the inputs, and have the processing done in pipe at execution time.

If the data is mandated to be copied, then it puts a penalty on implementation that could process the inputs directly in pipe. If the data is “referenced”, then it allows both types of implementation

The inputs are “referenced”, and should not be modified after the call to `vkCmdProcessCommandsNVX` and until after the rendering of the target command buffer is finished.

18) Why is this NVX and not NV?

To allow early experimentation and feedback. We expect that a version with a refined design as multi-vendor variant will follow up.

19) Should we make the availability for each token type a device limit?

Only distinguish between graphics/compute for now, further splitting up may lead to too much

fractioning.

20) When can the `objectTable` be modified?

Similar to the other inputs for `vkCmdProcessCommandsNVX`, only when all device access via `vkCmdProcessCommandsNVX` or execution of target command buffer has completed can an object at a given `objectIndex` be unregistered or re-registered again.

21) Which buffer usage flags are required for the buffers referenced by `vkCmdProcessCommandsNVX`

reuse existing `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT`

- `VkCmdProcessCommandsInfoNVX::sequencesCountBuffer`
- `VkCmdProcessCommandsInfoNVX::sequencesIndexBuffer`
- `VkIndirectCommandsTokenNVX::buffer`

22) In which pipeline stage do the device generated command expansion happen?

`vkCmdProcessCommandsNVX` is treated as if it occurs in a separate logical pipeline from either graphics or compute, and that pipeline only includes `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`, a new stage `VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX`, and `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`. This new stage has two corresponding new access types, `VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX` and `VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX`, used to synchronize reading the buffer inputs and writing the command buffer memory output. The output written in the target command buffer is considered to be consumed by the `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT` pipeline stage.

Thus, to synchronize from writing the input buffers to executing `vkCmdProcessCommandsNVX`, use:

- `dstStageMask = VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX`
- `dstAccessMask = VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX`

To synchronize from executing `vkCmdProcessCommandsNVX` to executing the generated commands, use

- `srcStageMask = VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX`
- `srcAccessMask = VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX`
- `dstStageMask = VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`
- `dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT`

When `vkCmdProcessCommandsNVX` is used with a `targetCommandBuffer` of `NULL`, the generated commands are immediately executed and there is implicit synchronization between generation and execution.

23) What if most token data is “static”, but we frequently want to render a subsection?

added “`sequencesIndexBuffer`”. This allows to easier sort and filter what should actually be processed.

## Example Code

Open-Source samples illustrating the usage of the extension can be found at the following locations:

[https://github.com/nvpro-samples/gl\\_vk\\_threaded\\_cadscene/blob/master/doc/vulkan\\_nvxdirectgenerated.md](https://github.com/nvpro-samples/gl_vk_threaded_cadscene/blob/master/doc/vulkan_nvxdirectgenerated.md)

<https://github.com/NVIDIAGameWorks/GraphicsSamples/tree/master/samples/vk10-kepler/BasicDeviceGeneratedCommandsVk>

```
// setup secondary command buffer
    vkBeginCommandBuffer(generatedCmdBuffer, &beginInfo);
    ... setup its state as usual

    // insert the reservation (there can only be one per command buffer)
    // where the generated calls should be filled into
    VkCmdReserveSpaceForCommandsInfoNVX reserveInfo = {
        VK_STRUCTURE_TYPE_CMD_RESERVE_SPACE_FOR_COMMANDS_INFO_NVX };
    reserveInfo.objectTable = objectTable;
    reserveInfo.indirectCommandsLayout = deviceGeneratedLayout;
    reserveInfo.maxSequencesCount = myCount;
    vkCmdReserveSpaceForCommandsNVX(generatedCmdBuffer, &reserveInfo);

    vkEndCommandBuffer(generatedCmdBuffer);

    // trigger the generation at some point in another primary command buffer
    VkCmdProcessCommandsInfoNVX processInfo = {
        VK_STRUCTURE_TYPE_CMD_PROCESS_COMMANDS_INFO_NVX };
    processInfo.objectTable = objectTable;
    processInfo.indirectCommandsLayout = deviceGeneratedLayout;
    processInfo.maxSequencesCount = myCount;
    // set the target of the generation (if null we would directly execute with
    mainCmd)
    processInfo.targetCommandBuffer = generatedCmdBuffer;
    // provide input data
    processInfo.indirectCommandsTokenCount = 3;
    processInfo.pIndirectCommandsTokens = myTokens;

    // If you modify the input buffer data referenced by VkCmdProcessCommandsInfoNVX,
    // ensure you have added the appropriate barriers prior generation process.
    // When regenerating the content of the same reserved space, ensure prior operations
    have completed

    VkMemoryBarrier memoryBarrier = { VK_STRUCTURE_TYPE_MEMORY_BARRIER };
    memoryBarrier.srcAccessMask = ...;
    memoryBarrier.dstAccessMask = VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX;

    vkCmdPipelineBarrier(mainCmd,
        /*srcStageMask*/VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
        /*dstStageMask*/VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX,
        /*dependencyFlags*/0,
```

```

        /*memoryBarrierCount*/1,
        /*pMemoryBarriers*/&memoryBarrier,
        ...);

vkCmdProcessCommandsNVX(mainCmd, &processInfo);
...

// execute the secondary command buffer and ensure the processing that modifies
command-buffer content
// has completed

memoryBarrier.srcAccessMask = VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX;
memoryBarrier.dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT;

vkCmdPipelineBarrier(mainCmd,
                     /*srcStageMask*/VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX,
                     /*dstStageMask*/VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT,
                     /*dependencyFlags*/0,
                     /*memoryBarrierCount*/1,
                     /*pMemoryBarriers*/&memoryBarrier,
                     ...)

vkCmdExecuteCommands(mainCmd, 1, &generatedCmdBuffer);

```

## Version History

- Revision 3, 2017-07-25 (Chris Hebert)
  - Correction to specification of dynamicCount for push\_constant token in VkIndirectCommandsLayoutNVX. Stride was incorrectly computed as dynamicCount was not treated as byte size.
- Revision 2, 2017-06-01 (Christoph Kubisch)
  - header compatibility break: add missing \_TYPE to VkIndirectCommandsTokenTypeNVX and VkObjectTypeNVX enums to follow Vulkan naming convention
  - behavior clarification: only allow a single work provoking token per sequence when creating a [VkIndirectCommandsLayoutNVX](#)
- Revision 1, 2016-10-31 (Christoph Kubisch)
  - Initial draft

## VK\_NVX\_image\_view\_handle

### Name String

`VK_NVX_image_view_handle`

### Extension Type

Device extension

### Registered Extension Number

31

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Eric Werness [@ewerness](#)

## Last Modified Date

2018-12-07

## Contributors

- Eric Werness, NVIDIA
- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA

This extension allows applications to query an opaque handle from an image view for use as a sampled image or storage image. This provides no direct functionality itself.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_IMAGE_VIEW_HANDLE_INFO_NVX`

## New Enums

None.

## New Structures

- [VkImageViewHandleInfoNVX](#)

## New Functions

- [vkGetImageViewHandleNVX](#)

## Issues

None.

## Version History

- Revision 1, 2018-12-07 (Eric Werness)

- Internal revisions

## **VK\_NVX\_multiview\_per\_view\_attributes**

### **Name String**

[VK\\_NVX\\_multiview\\_per\\_view\\_attributes](#)

### **Extension Type**

Device extension

### **Registered Extension Number**

98

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_multiview](#)

### **Contact**

- Jeff Bolz [@jeffbolz\\_nv](#)

### **Last Modified Date**

2017-01-13

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

- This extension requires the [SPV\\_NVX\\_multiview\\_per\\_view\\_attributes](#) SPIR-V extension.
- This extension requires the [GL\\_NVX\\_multiview\\_per\\_view\\_attributes](#) extension for GLSL source languages.
- This extension interacts with [VK\\_NV\\_viewport\\_array2](#).

### **Contributors**

- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA

This extension adds a new way to write shaders to be used with multiview subpasses, where the attributes for all views are written out by a single invocation of the vertex processing stages. Related SPIR-V and GLSL extensions [SPV\\_NVX\\_multiview\\_per\\_view\\_attributes](#) and [GL\\_NVX\\_multiview\\_per\\_view\\_attributes](#) introduce per-view position and viewport mask attributes arrays, and this extension defines how those per-view attribute arrays are interpreted by Vulkan. Pipelines using per-view attributes **may** only execute the vertex processing stages once for all views rather than once per-view, which reduces redundant shading work.

A subpass creation flag controls whether the subpass uses this extension. A subpass **must** either exclusively use this extension or not use it at all.

Some Vulkan implementations only support the position attribute varying between views in the X component. A subpass can declare via a second creation flag whether all pipelines compiled for this subpass will obey this restriction.

Shaders that use the new per-view outputs (e.g. `gl_PositionPerViewNV`) **must** also write the non-per-view output (`gl_Position`), and the values written **must** be such that `gl_Position = gl_PositionPerViewNV[gl_ViewIndex]` for all views in the subpass. Implementations are free to either use the per-view outputs or the non-per-view outputs, whichever would be more efficient.

If `VK_NV_viewport_array2` is not also supported and enabled, the per-view viewport mask **must** not be used.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PER_VIEW_ATTRIBUTES_PROPERTIES_NVX`
- Extending `VkSubpassDescriptionFlagBits`
  - `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX`
  - `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX`

## New Functions

None.

## New Built-In Variables

- `PositionPerViewNV`
- `ViewportMaskPerViewNV`

## New SPIR-V Capabilities

- `PerViewAttributesNV`

## Issues

None.

## Examples

```
#version 450 core

#extension GL_KHX_multiview : enable
#extension GL_NVX_multiview_per_view_attributes : enable

layout(location = 0) in vec4 position;
layout(set = 0, binding = 0) uniform Block { mat4 mvpPerView[2]; } buf;

void main()
{
    // Output both per-view positions and gl_Position as a function
    // of gl_ViewIndex
    gl_PositionPerViewNV[0] = buf.mvpPerView[0] * position;
    gl_PositionPerViewNV[1] = buf.mvpPerView[1] * position;
    gl_Position = buf.mvpPerView[gl_ViewIndex] * position;
}
```

## Version History

- Revision 1, 2017-01-13 (Jeff Bolz)
  - Internal revisions

## VK\_NV\_clip\_space\_w\_scaling

### Name String

VK\_NV\_clip\_space\_w\_scaling

### Extension Type

Device extension

### Registered Extension Number

88

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Eric Werness [@ewerness-nv](#)

## Last Modified Date

2017-02-15

## Contributors

- Eric Werness, NVIDIA
- Kedarnath Thangudu, NVIDIA

Virtual Reality (VR) applications often involve a post-processing step to apply a “barrel” distortion to the rendered image to correct the “pincushion” distortion introduced by the optics in a VR device. The barrel distorted image has lower resolution along the edges compared to the center. Since the original image is rendered at high resolution, which is uniform across the complete image, a lot of pixels towards the edges do not make it to the final post-processed image.

This extension provides a mechanism to render VR scenes at a non-uniform resolution, in particular a resolution that falls linearly from the center towards the edges. This is achieved by scaling the w coordinate of the vertices in the clip space before perspective divide. The clip space w coordinate of the vertices **can** be offset as of a function of x and y coordinates as follows:

$$w' = w + Ax + By$$

In the intended use case for viewport position scaling, an application should use a set of four viewports, one for each of the four quadrants of a Cartesian coordinate system. Each viewport is set to the dimension of the image, but is scissored to the quadrant it represents. The application should specify A and B coefficients of the w-scaling equation above, that have the same value, but different signs, for each of the viewports. The signs of A and B should match the signs of x and y for the quadrant that they represent such that the value of  $w'$  will always be greater than or equal to the original w value for the entire image. Since the offset to w, ( $Ax + By$ ), is always positive, and increases with the absolute values of x and y, the effective resolution will fall off linearly from the center of the image to its edges.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV`
- Extending [VkDynamicState](#):
  - `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV`

## New Enums

None.

## New Structures

- [VkViewportWScalingNV](#)

- [VkPipelineViewportWScalingStateCreateInfoNV](#)

## New Functions

- [vkCmdSetViewportWScalingNV](#)

## Issues

1) Is the pipeline struct name too long?

**RESOLVED:** It fits with the naming convention.

2) Separate W scaling section or fold into coordinate transformations?

**RESOLVED:** Leaving it as its own section for now.

## Examples

```

VkViewport viewports[4];
VkRect2D scissors[4];
VkViewportWScalingNV scalings[4];

for (int i = 0; i < 4; i++) {
    int x = (i & 2) ? 0 : currentWindowWidth / 2;
    int y = (i & 1) ? 0 : currentWindowHeight / 2;

    viewports[i].x = 0;
    viewports[i].y = 0;
    viewports[i].width = currentWindowWidth;
    viewports[i].height = currentWindowHeight;
    viewports[i].minDepth = 0.0f;
    viewports[i].maxDepth = 1.0f;

    scissors[i].offset.x = x;
    scissors[i].offset.y = y;
    scissors[i].extent.width = currentWindowWidth/2;
    scissors[i].extent.height = currentWindowHeight/2;

    const float factor = 0.15;
    scalings[i].xcoeff = ((i & 2) ? -1.0 : 1.0) * factor;
    scalings[i].ycoeff = ((i & 1) ? -1.0 : 1.0) * factor;
}

```

```

VKPipelineViewportWScalingStateCreateInfoNV vpWScalingStateInfo = {
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV };

```

```

vpWScalingStateInfo.viewportWScalingEnable = VK_TRUE;
vpWScalingStateInfo.viewportCount = 4;
vpWScalingStateInfo.pViewportWScalings = &scalings[0];

```

```

VKPipelineViewStateCreateInfo vpStateInfo = {
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO };
vpStateInfo.viewportCount = 4;
vpStateInfo.pViewports = &viewports[0];
vpStateInfo.scissorCount = 4;
vpStateInfo.pScissors = &scissors[0];
vpStateInfo.pNext = &vpWScalingStateInfo;

```

Example shader to read from a w-scaled texture:

```

// Vertex Shader
// Draw a triangle that covers the whole screen
const vec4 positions[3] = vec4[3](vec4(-1, -1, 0, 1),
                                vec4( 3, -1, 0, 1),
                                vec4(-1, 3, 0, 1));
out vec2 uv;
void main()
{
    vec4 pos = positions[ gl_VertexID ];
    gl_Position = pos;
    uv = pos.xy;
}

// Fragment Shader
uniform sampler2D tex;
uniform float xcoeff;
uniform float ycoeff;
out vec4 Color;
in vec2 uv;

void main()
{
    // Handle uv as if upper right quadrant
    vec2 uvabs = abs(uv);

    // unscale: transform w-scaled image into an unscaled image
    // scale: transform unscaled image int a w-scaled image
    float unscale = 1.0 / (1 + xcoeff * uvabs.x + xcoeff * uvabs.y);
    //float scale = 1.0 / (1 - xcoeff * uvabs.x - xcoeff * uvabs.y);

    vec2 P = vec2(unscale * uvabs.x, unscale * uvabs.y);

    // Go back to the right quadrant
    P *= sign(uv);

    Color = texture(tex, P * 0.5 + 0.5);
}

```

## Version History

- Revision 1, 2017-02-15 (Eric Werness)
  - Internal revisions

## VK\_NV\_compute\_shader\_derivatives

### Name String

`VK_NV_compute_shader_derivatives`

## Extension Type

Device extension

## Registered Extension Number

202

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

## Contact

- Pat Brown [Onvpbrown](#)

## Last Modified Date

2018-07-19

## IP Status

No known IP claims.

## Contributors

- Pat Brown, NVIDIA

This extension adds Vulkan support for the `SPV_NV_compute_shader_derivatives` SPIR-V extension.

The SPIR-V extension provides two new execution modes, both of which allow compute shaders to use built-ins that evaluate compute derivatives explicitly or implicitly. Derivatives will be computed via differencing over a 2x2 group of shader invocations. The `DerivativeGroupQuadsNV` execution mode assembles shader invocations into 2x2 groups, where each group has x and y coordinates of the local invocation ID of the form  $(2m+\{0,1\}, 2n+\{0,1\})$ . The `DerivativeGroupLinearNV` execution mode assembles shader invocations into 2x2 groups, where each group has local invocation index values of the form  $4m+\{0,1,2,3\}$ .

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COMPUTE_SHADER_DERIVATIVES_FEATURES_NV`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceComputeShaderDerivativesFeaturesNV](#)

## New Functions

None.

## New SPIR-V Capability

- [ComputeDerivativeGroupQuadsNV](#)
- [ComputeDerivativeGroupLinearNV](#)

## Issues

(1) Should we specify that the groups of four shader invocations used for derivatives in a compute shader are the same groups of four invocations that form a “quad” in shader subgroups?

**RESOLVED:** Yes.

## Examples

None.

## Version History

- Revision 1, 2018-07-19 (Pat Brown)
  - Initial draft

## VK\_NV\_cooperative\_matrix

### Name String

`VK_NV_cooperative_matrix`

### Extension Type

Device extension

### Registered Extension Number

250

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Jeff Bolz [@jeffbolz](#)

## Last Modified Data

2019-02-05

## Contributors

- Jeff Bolz, NVIDIA
- Markus Tavenrath, NVIDIA
- Daniel Koch, NVIDIA

This extension adds support for using cooperative matrix types in SPIR-V. Cooperative matrix types are medium-sized matrices that are primarily supported in compute shaders, where the storage for the matrix is spread across all invocations in some scope (usually a subgroup) and those invocations cooperate to efficiently perform matrix multiplies.

Cooperative matrix types are defined by the `SPV_NV_cooperative_matrix` SPIR-V extension and can be used with the `GL_NV_cooperative_matrix` GLSL extension.

This extension includes support for enumerating the matrix types and dimensions that are supported by the implementation.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_COOPERATIVE_MATRIX_PROPERTIES_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_PROPERTIES_NV`

## New Enums

- `VkScopeNV`
- `VkComponentTypeNV`

## New Structures

- Extending `VkPhysicalDeviceFeatures2`:
  - `VkPhysicalDeviceCooperativeMatrixFeaturesNV`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceCooperativeMatrixPropertiesNV`
- `VkCooperativeMatrixPropertiesNV`

## New Functions

- [vkGetPhysicalDeviceCooperativeMatrixPropertiesNV](#)

## New SPIR-V Capabilities

- [CooperativeMatrixNV](#)

## Issues

(1) What matrix properties will be supported in practice?

RESOLVED: In NVIDIA's initial implementation, we will support:

- AType = BType = fp16 CType = DType = fp16 MxNxK = 16x8x16 scope = Subgroup
- AType = BType = fp16 CType = DType = fp16 MxNxK = 16x8x8 scope = Subgroup
- AType = BType = fp16 CType = DType = fp32 MxNxK = 16x8x16 scope = Subgroup
- AType = BType = fp16 CType = DType = fp32 MxNxK = 16x8x8 scope = Subgroup

## Version History

- Revision 1, 2019-02-05 (Jeff Bolz)
  - Internal revisions

## VK\_NV\_corner\_sampled\_image

### Name String

`VK_NV_corner_sampled_image`

### Extension Type

Device extension

### Registered Extension Number

51

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Daniel Koch [dgkoch](#)

### Last Modified Date

2018-08-13

## Contributors

- Jeff Bolz, NVIDIA
- Pat Brown, NVIDIA
- Chris Lentini, NVIDIA

This extension adds support for a new image organization, which this extension refers to as “corner-sampled” images. A corner-sampled image differs from a conventional image in the following ways:

- Texels are centered on integer coordinates. See [Unnormalized Texel Coordinate Operations](#)
- Normalized coordinates are scaled using  $\text{coord} * (\text{dim} - 1)$  rather than  $\text{coord} * \text{dim}$ , where  $\text{dim}$  is the size of one dimension of the image. See [normalized texel coordinate transform](#).
- Partial derivatives are scaled using  $\text{coord} * (\text{dim} - 1)$  rather than  $\text{coord} * \text{dim}$ . See [Scale Factor Operation](#).
- Calculation of the next higher lod size goes according to  $\text{dim} / 2$  rather than  $\text{dim} / 2$ . See [Image Miplevel Sizing](#).
- The minimum level size is 2x2 for 2D images and 2x2x2 for 3D images. See [Image Miplevel Sizing](#).

This image organization is designed to facilitate a system like Ptex with separate textures for each face of a subdivision or polygon mesh. Placing sample locations at pixel corners allows applications to maintain continuity between adjacent patches by duplicating values along shared edges. Additionally, using the modified mipmapping logic along with texture dimensions of the form  $2^n + 1$  allows continuity across shared edges even if the adjacent patches use different level-of-detail values.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#)
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CORNER_SAMPLED_IMAGE_FEATURES_NV`
- Extending [VkImageCreateFlagBits](#)
  - `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceCornerSampledImageFeaturesNV](#)

## New Functions

None.

## New Built-In Variables

None.

## New SPIR-V Capabilities

None.

## Issues

1. What should this extension be named?

DISCUSSION: While naming this extension, we chose the most distinctive aspect of the image organization and referred to such images as “corner-sampled images”. As a result, we decided to name the extension NV\_corner\_sampled\_image.

2. Do we need a format feature flag so formats can advertise if they support corner-sampling?

DISCUSSION: Currently NVIDIA supports this for all 2D and 3D formats, but not for cubemaps or depth-stencil formats. A format feature might be useful if other vendors would only support this on some formats.

3. Do integer texel coordinates have a different range for corner-sampled images?

RESOLVED: No, these are unchanged.

4. Do unnormalized sampler coordinates work with corner-sampled images? Are there any functional differences?

RESOLVED: Yes they work. Unnormalized coordinates are treated as already scaled for corner-sample usage.

5. Should we have a diagram in the “Image Operations” chapter demonstrating different texel sampling locations?

UNRESOLVED: Probably, but later.

## Version History

- Revision 1, 2018-08-14 (Daniel Koch)
  - Internal revisions
- Revision 2, 2018-08-14 (Daniel Koch)
  - ???

## **VK\_NV\_coverage\_reduction\_mode**

### **Name String**

`VK_NV_coverage_reduction_mode`

### **Extension Type**

Device extension

### **Registered Extension Number**

251

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_NV_framebuffer_mixed_samples`

### **Contact**

- Kedarnath Thangudu [Okthangudu](#)

### **Last Modified Date**

2019-01-29

### **Contributors**

- Kedarnath Thangudu, NVIDIA
- Jeff Bolz, NVIDIA

When using a framebuffer with mixed samples, a per-fragment coverage reduction operation is performed which generates a color sample mask from the coverage mask. This extension defines the following modes to control how this reduction is performed.

- Merge: When there are more raster samples than color samples, there is an implementation dependent association of each raster sample to a color sample. In the merge mode, the reduced color sample mask is computed such that the bit for each color sample is 1 if any of the associated bits in the fragment's coverage is on, and 0 otherwise. This is the default mode.
- Truncate: When there are more raster samples (N) than color samples(M), there is one to one association of the first M raster samples to the M color samples and the coverage bits for the other raster samples are ignored.

When the number of raster samples is equal to the color samples, there is a one to one mapping between them in either of the above modes.

The new command `vkGetPhysicalDeviceSupportedFramebuffersMixedSamplesCombinationsNV` can be used to query the various raster, color, depth/stencil sample count and reduction mode combinations that are supported by the implementation. This extension would allow an implementation to support the behavior of both `VK_NV_framebuffer_mixed_samples` and `VK_AMD_mixed_attachment_samples` extensions simultaneously.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#)
  - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_COVERAGE\_REDUCTION\_MODE\_FEATURES\_NV
  - VK\_STRUCTURE\_TYPE\_PIPELINE\_COVERAGE\_REDUCTION\_STATE\_CREATE\_INFO\_NV
  - VK\_STRUCTURE\_TYPE\_FRAMEBUFFER\_MIXED\_SAMPLES\_COMBINATION\_NV

## New Enums

- [VkCoverageReductionModeNV](#)
- [VkPipelineCoverageReductionStateCreateInfoNV](#)

## New Structures

- [VkPhysicalDeviceCoverageReductionModeFeaturesNV](#)
- [VkPipelineCoverageReductionStateCreateInfoNV](#)
- [VkFramebufferMixedSamplesCombinationNV](#)

## New Functions

- [vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV](#)

## Issues

None.

## Version History

- Revision 1, 2019-01-29 (Kedarnath Thangudu)
  - Internal revisions

## VK\_NV\_dedicated\_allocation\_image\_aliasing

### Name String

VK\_NV\_dedicated\_allocation\_image\_aliasing

### Extension Type

Device extension

### Registered Extension Number

241

### Revision

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_dedicated\\_allocation](#)

## Contact

- Nuno Subtil [@nsubtil](#)

## Last Modified Date

2019-01-04

## Contributors

- Nuno Subtil, NVIDIA
- Jeff Bolz, NVIDIA
- Eric Werness, NVIDIA
- Axel Gneiting, id Software

This extension allows applications to alias images on dedicated allocations, subject to specific restrictions: the extent and the number of layers in the image being aliased must be smaller than or equal to those of the original image for which the allocation was created, and every other image parameter must match.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#)
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEDICATED_ALLOCATION_IMAGE_ALIASING_FEATURES_NV`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2019-01-04 (Nuno Subtil)
  - Internal revisions

## VK\_NV\_device\_diagnostic\_checkpoints

### Name String

`VK_NV_device_diagnostic_checkpoints`

### Extension Type

Device extension

### Registered Extension Number

207

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Nuno Subtil [Onsubtil](#)

### Last Modified Date

2018-07-16

### Contributors

- Oleg Kuznetsov, NVIDIA
- Alex Dunn, NVIDIA
- Jeff Bolz, NVIDIA
- Eric Werness, NVIDIA
- Daniel Koch, NVIDIA

This extension allows applications to insert markers in the command stream and associate them with custom data.

If a device lost error occurs, the application **may** then query the implementation for the last markers to cross specific implementation-defined pipeline stages, in order to narrow down which commands were executing at the time and might have caused the failure.

## New Object Types

None.

## New Enum Constants

Extending [VkStructureType](#):

- [VK\\_STRUCTURE\\_TYPE\\_CHECKPOINT\\_DATA\\_NV](#)
- [VK\\_STRUCTURE\\_TYPE\\_QUEUE\\_FAMILY\\_CHECKPOINT\\_PROPERTIES\\_NV](#)

## New Enums

None.

## New Structures

- [VkCheckpointDataNV](#)
- [VkQueueFamilyCheckpointPropertiesNV](#)

## New Functions

- [vkCmdSetCheckpointNV](#)
- [vkGetQueueCheckpointDataNV](#)

## Issues

None yet!

## Version History

- Revision 1, 2018-07-16 (Nuno Subtil)
  - Internal revisions
- Revision 2, 2018-07-16 (Nuno Subtil)
  - ???

## VK\_NV\_fill\_rectangle

### Name String

[VK\\_NV\\_fill\\_rectangle](#)

### Extension Type

Device extension

### Registered Extension Number

154

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Jeff Bolz [@jeffbolz\\_nv](#)

## Last Modified Date

2017-05-22

## Contributors

- Jeff Bolz, NVIDIA

This extension adds a new `VkPolygonMode` enum where a triangle is rasterized by computing and filling its axis-aligned screen-space bounding box, disregarding the actual triangle edges. This can be useful for drawing a rectangle without being split into two triangles with an internal edge. It is also useful to minimize the number of primitives that need to be drawn, particularly for a user interface.

## New Object Types

None.

## New Enum Constants

- Extending `VkPolygonMode`
  - `VK_POLYGON_MODE_FILL_RECTANGLE_NV`

## New Enums

None.

## New Structures

None.

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2017-05-22 (Jeff Bolz)
  - Internal revisions

## VK\_NV\_fragment\_coverage\_to\_color

### Name String

`VK_NV_fragment_coverage_to_color`

### Extension Type

Device extension

### Registered Extension Number

150

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Jeff Bolz [@jeffbolznv](#)

### Last Modified Date

2017-05-21

### Contributors

- Jeff Bolz, NVIDIA

This extension allows the fragment coverage value, represented as an integer bitmask, to be substituted for a color output being written to a single-component color attachment with integer components (e.g. `VK_FORMAT_R8_UINT`). The functionality provided by this extension is different from simply writing the `SampleMask` fragment shader output, in that the coverage value written to the framebuffer is taken after stencil test and depth test, as well as after fragment operations such as alpha-to-coverage.

This functionality may be useful for deferred rendering algorithms, where the second pass needs to know which samples belong to which original fragments.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_TO_COLOR_STATE_CREATE_INFO_NV`

## New Enums

- `VkPipelineCoverageToColorStateCreateFlagsNV`

## New Structures

- [VkPipelineCoverageToColorStateCreateInfoNV](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2017-05-21 (Jeff Bolz)
  - Internal revisions

## VK\_NV\_fragment\_shader\_barycentric

### Name String

`VK_NV_fragment_shader_barycentric`

### Extension Type

Device extension

### Registered Extension Number

204

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Pat Brown [Onvpbrown](#)

### Last Modified Date

2018-08-03

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Requires the `SPV_NV_fragment_shader_barycentric` SPIR-V extension.
- Requires the `GL_NV_fragment_shader_barycentric` extension for GLSL source languages.

## Contributors

- Pat Brown, NVIDIA
- Daniel Koch, NVIDIA

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_NV_fragment_shader_barycentric`

The extension provides access to three additional fragment shader variable decorations in SPIR-V:

- `PerVertexNV`, which indicates that a fragment shader input will not have interpolated values, but instead must be accessed with an extra array index that identifies one of the vertices of the primitive producing the fragment
- `BaryCoordNV`, which indicates that the variable is a three-component floating-point vector holding barycentric weights for the fragment produced using perspective interpolation
- `BaryCoordNoPerspNV`, which indicates that the variable is a three-component floating-point vector holding barycentric weights for the fragment produced using linear interpolation

When using GLSL source-based shader languages, the following variables from `GL_NV_fragment_shader_barycentric` maps to these SPIR-V built-in decorations:

- `in vec3 gl_BaryCoordNV; → BaryCoordNV`
- `in vec3 gl_BaryCoordNoPerspNV; → BaryCoordNoPerspNV`

GLSL variables declared using the `_pervertexNV` GLSL qualifier are expected to be decorated with `PerVertexNV` in SPIR-V.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_BARYCENTRIC_FEATURES_NV`

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New Built-In Variables

- `BaryCoordNV`
- `BaryCoordNoPerspNV`

## New SPIR-V Decorations

- `PerVertexNV`

## New SPIR-V Capabilities

- `FragmentBarycentricNV`

## Issues

(1) The AMD\_shader\_explicit\_vertex\_parameter extension provides similar functionality. Why write a new extension, and how is this extension different?

**RESOLVED:** For the purposes of Vulkan/SPIR-V, we chose to implement a separate extension due to several functional differences.

First, the hardware supporting this extension can provide a three-component barycentric weight vector for variables decorated with `BaryCoordNV`, while variables decorated with `BaryCoordSmoothAMD` provide only two components. In some cases, it may be more efficient to explicitly interpolate an attribute via:

```
float value = (baryCoordNV.x * v[0].attrib +  
               baryCoordNV.y * v[1].attrib +  
               baryCoordNV.z * v[2].attrib);
```

instead of

```
float value = (baryCoordSmoothAMD.x * (v[0].attrib - v[2].attrib) +  
               baryCoordSmoothAMD.y * (v[1].attrib - v[2].attrib) +  
               v[2].attrib);
```

Additionally, the semantics of the decoration `BaryCoordPullModelAMD` do not appear to map to anything supported by the initial hardware implementation of this extension.

This extension provides a smaller number of decorations than the AMD extension, as we expect that shaders could derive variables decorated with things like `BaryCoordNoPerspCentroidAMD` with explicit attribute interpolation instructions. One other relevant difference is that explicit per-vertex attribute access using this extension does not require a constant vertex number.

(2) Why do the built-in SPIR-V decorations for this extension include two separate built-ins `BaryCoordNV` and `BaryCoordNoPerspNV` when a “no perspective” variable could be decorated with `BaryCoordNV` and `NoPerspective`?

**RESOLVED:** The SPIR-V extension for this feature chose to mirror the behavior of the GLSL extension, which provides two built-in variables. Additionally, it's not clear that it's a good idea (or even legal) to have two variables using the “same attribute”, but with different interpolation modifiers.

## Version History

- Revision 1, 2018-08-03 (Pat Brown)
  - Internal revisions

## **VK\_NV\_framebuffer\_mixed\_samples**

### Name String

`VK_NV_framebuffer_mixed_samples`

### Extension Type

Device extension

### Registered Extension Number

153

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Jeff Bolz [@jeffbolz\\_nv](#)

### Last Modified Date

2017-06-04

### Contributors

- Jeff Bolz, NVIDIA

This extension allows multisample rendering with a raster and depth/stencil sample count that is larger than the color sample count. Rasterization and the results of the depth and stencil tests together determine the portion of a pixel that is “covered”. It can be useful to evaluate coverage at a higher frequency than color samples are stored. This coverage is then “reduced” to a collection of covered color samples, each having an opacity value corresponding to the fraction of the color sample covered. The opacity can optionally be blended into individual color samples.

Rendering with fewer color samples than depth/stencil samples greatly reduces the amount of memory and bandwidth consumed by the color buffer. However, converting the coverage values into opacity introduces artifacts where triangles share edges and **may** not be suitable for normal triangle mesh rendering.

One expected use case for this functionality is Stencil-then-Cover path rendering (similar to the

OpenGL GL\_NV\_path\_rendering extension). The stencil step determines the coverage (in the stencil buffer) for an entire path at the higher sample frequency, and then the cover step draws the path into the lower frequency color buffer using the coverage information to antialias path edges. With this two-step process, internal edges are fully covered when antialiasing is applied and there is no corruption on these edges.

The key features of this extension are:

- It allows render pass and framebuffer objects to be created where the number of samples in the depth/stencil attachment in a subpass is a multiple of the number of samples in the color attachments in the subpass.
- A coverage reduction step is added to Fragment Operations which converts a set of covered raster/depth/stencil samples to a set of color samples that perform blending and color writes. The coverage reduction step also includes an optional coverage modulation step, multiplying color values by a fractional opacity corresponding to the number of associated raster/depth/stencil samples covered.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#)
  - [VK\\_STRUCTURE\\_TYPE\\_PIPELINE\\_COVERAGE\\_MODULATION\\_STATE\\_CREATE\\_INFO\\_NV](#)

## New Enums

- [VkCoverageModulationModeNV](#)
- [VkPipelineCoverageModulationStateCreateFlagsNV](#)

## New Structures

- [VkPipelineCoverageModulationStateCreateInfoNV](#)

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2017-06-04 (Jeff Bolz)
  - Internal revisions

# **VK\_NV\_geometry\_shader\_passthrough**

## **Name String**

`VK_NV_geometry_shader_passthrough`

## **Extension Type**

Device extension

## **Registered Extension Number**

96

## **Revision**

1

## **Extension and Version Dependencies**

- Requires Vulkan 1.0

## **Contact**

- Daniel Koch [Qdgkoch](#)

## **Last Modified Date**

2017-02-15

## **Interactions and External Dependencies**

- This extension requires the `SPV_NV_geometry_shader_passthrough` SPIR-V extension.
- This extension requires the `GL_NV_geometry_shader_passthrough` extension for GLSL source languages.
- This extension requires the `geometryShader` feature.

## **Contributors**

- Piers Daniell, NVIDIA
- Jeff Bolz, NVIDIA

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_NV_geometry_shader_passthrough`

Geometry shaders provide the ability for applications to process each primitive sent through the graphics pipeline using a programmable shader. However, one common use case treats them largely as a “passthrough”. In this use case, the bulk of the geometry shader code simply copies inputs from each vertex of the input primitive to corresponding outputs in the vertices of the output primitive. Such shaders might also compute values for additional built-in or user-defined per-primitive attributes (e.g., `Layer`) to be assigned to all the vertices of the output primitive.

This extension provides access to the `PassthroughNV` decoration under the `GeometryShaderPassthroughNV` capability. Adding this to a geometry shader input variable specifies that the values of this input are copied to the corresponding vertex of the output primitive.

When using GLSL source-based shading languages, the `passthrough` layout qualifier from `GL_NV_geometry_shader_passthrough` maps to the `PassthroughNV` decoration. To use the `passthrough` layout, in GLSL the `GL_NV_geometry_shader_passthrough` extension must be enabled. Behaviour is described in the `GL_NV_geometry_shader_passthrough` extension specification.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New Built-In Variables

None.

## New Variable Decoration

- `PassthroughNV` in [Geometry Shader Passthrough](#)

## New SPIR-V Capabilities

- [GeometryShaderPassthroughNV](#)

## Issues

1) Should we require or allow a passthrough geometry shader to specify the output layout qualifiers for the output primitive type and maximum vertex count in the SPIR-V?

**RESOLVED:** Yes they should be required in the SPIR-V. Per `GL_NV_geometry_shader_passthrough` they are not permitted in the GLSL source shader, but SPIR-V is lower-level. It is straightforward for the GLSL compiler to infer them from the input primitive type and to explicitly emit them in the SPIR-V according to the following table.

Input Layout	Implied Output Layout
points	<code>layout(points, max_vertices=1)</code>

Input Layout	Implied Output Layout
lines	<code>layout(line_strip, max_vertices=2)</code>
triangles	<code>layout(triangle_strip, max_vertices=3)</code>

2) How does interface matching work with passthrough geometry shaders?

**RESOLVED:** This is described in [Passthrough Interface Matching](#). In GL when using passthrough geometry shaders in separable mode, all inputs must also be explicitly assigned location layout qualifiers. In Vulkan all SPIR-V shader inputs (except built-ins) must also have location decorations specified. Redeclarations of built-in variables that add the passthrough layout qualifier are exempted from the rule requiring location assignment because built-in variables do not have locations and are matched by `BuiltIn` decoration.

## Sample Code

Consider the following simple geometry shader in unextended GLSL:

```

layout(triangles) in;
layout(triangle_strip) out;
layout(max_vertices=3) out;

in Inputs {
    vec2 texcoord;
    vec4 baseColor;
} v_in[];
out Outputs {
    vec2 texcoord;
    vec4 baseColor;
};

void main()
{
    int layer = compute_layer();
    for (int i = 0; i < 3; i++) {
        gl_Position = gl_in[i].gl_Position;
        texcoord = v_in[i].texcoord;
        baseColor = v_in[i].baseColor;
        gl_Layer = layer;
        EmitVertex();
    }
}

```

In this shader, the inputs `gl_Position`, `Inputs.texcoord`, and `Inputs.baseColor` are simply copied from the input vertex to the corresponding output vertex. The only “interesting” work done by the geometry shader is computing and emitting a `gl_Layer` value for the primitive.

The following geometry shader, using this extension, is equivalent:

```
#extension GL_NV_geometry_shader_passthrough : require

layout(triangles) in;
// No output primitive layout qualifiers required.

// Redefine gl_PerVertex to pass through "gl_Position".
layout(passthrough) in gl_PerVertex {
    vec4 gl_Position;
} gl_in[];

// Declare "Inputs" with "passthrough" to automatically copy members.
layout(passthrough) in Inputs {
    vec2 texcoord;
    vec4 baseColor;
} v_in[];

// No output block declaration required.

void main()
{
    // The shader simply computes and writes gl_Layer. We don't
    // loop over three vertices or call EmitVertex().
    gl_Layer = compute_layer();
}
```

## Version History

- Revision 1, 2017-02-15 (Daniel Koch)
  - Internal revisions

## VK\_NV\_mesh\_shader

### Name String

`VK_NV_mesh_shader`

### Extension Type

Device extension

### Registered Extension Number

203

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

## Contact

- Christoph Kubisch [@pixeljetstream](#)

## Last Modified Date

2018-07-19

## Contributors

- Pat Brown, NVIDIA
- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA
- Piers Daniell, NVIDIA
- Pierre Boudier, NVIDIA

This extension provides a new mechanism allowing applications to generate collections of geometric primitives via programmable mesh shading. It is an alternative to the existing programmable primitive shading pipeline, which relied on generating input primitives by a fixed function assembler as well as fixed function vertex fetch.

There are new programmable shader types—the task and mesh shader—to generate these collections to be processed by fixed-function primitive assembly and rasterization logic. When the task and mesh shaders are dispatched, they replace the standard programmable vertex processing pipeline, including vertex array attribute fetching, vertex shader processing, tessellation, and the geometry shader processing.

This extension also adds support for the following SPIR-V extension in Vulkan:

- `SPV_NV_mesh_shader`

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_PROPERTIES_NV`
- Extending [VkShaderStageFlagBits](#)
  - `VK_SHADER_STAGE_TASK_BIT_NV`
  - `VK_SHADER_STAGE_MESH_BIT_NV`
- Extending [VkPipelineStageFlagBits](#)
  - `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`
  - `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceMeshShaderFeaturesNV](#)
- [VkPhysicalDeviceMeshShaderPropertiesNV](#)
- [VkDrawMeshTasksIndirectCommandNV](#)

## New Functions

- [vkCmdDrawMeshTasksNV](#)
- [vkCmdDrawMeshTasksIndirectNV](#)
- [vkCmdDrawMeshTasksIndirectCountNV](#)

## New or Modified Built-In Variables

- [TaskCountNV](#)
- [PrimitiveCountNV](#)
- [PrimitiveIndicesNV](#)
- [ClipDistancePerViewNV](#)
- [CullDistancePerViewNV](#)
- [LayerPerViewNV](#)
- [MeshViewCountNV](#)
- [MeshViewIndicesNV](#)
- (modified)[Position](#)
- (modified)[PointSize](#)
- (modified)[ClipDistance](#)
- (modified)[CullDistance](#)
- (modified)[PrimitiveId](#)
- (modified)[Layer](#)
- (modified)[ViewportIndex](#)
- (modified)[WorkgroupSize](#)
- (modified)[WorkgroupId](#)
- (modified)[LocalInvocationId](#)
- (modified)[GlobalInvocationId](#)
- (modified)[LocalInvocationIndex](#)
- (modified)[DrawIndex](#)

- (modified) `ViewportMaskNV`
- (modified) `PositionPerViewNV`
- (modified) `ViewportMaskPerViewNV`

## New SPIR-V Capability

- `MeshShadingNV`

## Issues

1. How to name this extension?

RESOLVED: `VK_NV_mesh_shader`

Other options considered:

- `VK_NV_mesh_shading`
- `VK_NV_programmable_mesh_shading`
- `VK_NV_primitive_group_shading`
- `VK_NV_grouped_drawing`

2. Do we need a new `VkPrimitiveTopology`?

RESOLVED: NO, we skip the `InputAssembler` stage

3. Should we allow Instancing?

RESOLVED: NO, there is no fixed function input, other than the IDs. However, allow offsetting with a "first" value.

4. Should we use existing `vkCmdDraw` or introduce new functions?

RESOLVED: Introduce new functions.

New functions make it easier to separate from "programmable primitive shading" chapter, less "dual use" language about existing functions having alternative behavior. The text around the existing "draws" is heavily based around emitting vertices.

5. If new functions, how to name?

RESOLVED: `CmdDrawMeshTasks*`

Other options considered:

- `CmdDrawMeshed`
- `CmdDrawTasked`
- `CmdDrawGrouped`

6. Should `VK_SHADER_STAGE_ALL_GRAPHICS` be updated to include the new stages?

**RESOLVED:** No. If an application were to be recompiled with headers that include additional shader stage bits in VK\_SHADER\_STAGE\_ALL\_GRAPHICS, then the previously valid application would no longer be valid on implementations that don't support mesh or task shaders. This means the change would not be backwards compatible. It's too bad VkShaderStageFlagBits doesn't have a dedicated "all supported graphics stages" bit like VK\_PIPELINE\_STAGE\_ALL\_GRAPHICS\_BIT, which would have avoided this problem.

## Version History

- Revision 1, 2018-07-19 (Christoph Kubisch, Daniel Koch)
  - Internal revisions

## VK\_NV\_ray\_tracing

### Name String

`VK_NV_ray_tracing`

### Extension Type

Device extension

### Registered Extension Number

166

### Revision

3

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_get_memory_requirements2`

### Contact

- Eric Werness [@ewerness](#)

### Last Modified Date

2018-11-20

### Interactions and External Dependencies

- This extension requires the `SPV_NV_ray_tracing` SPIR-V extension.
- This extension requires the `GL_NV_ray_tracing` extension for GLSL source languages.

### Contributors

- Eric Werness, NVIDIA
- Ashwin Lele, NVIDIA
- Robert Stepinski, NVIDIA

- Nuno Subtil, NVIDIA
- Christoph Kubisch, NVIDIA
- Martin Stich, NVIDIA
- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA
- Joshua Barczak, Intel
- Tobias Hector, AMD
- Henrik Rydgard, NVIDIA
- Pascal Gautron, NVIDIA

Rasterization has been the dominant method to produce interactive graphics, but increasing performance of graphics hardware has made ray tracing a viable option for interactive rendering. Being able to integrate ray tracing with traditional rasterization makes it easier for applications to incrementally add ray traced effects to existing applications or to do hybrid approaches with rasterization for primary visibility and ray tracing for secondary queries.

To enable ray tracing, this extension adds a few different categories of new functionality:

- Acceleration structure objects and build commands
- A new pipeline type with new shader domains
- An indirection table to link shader groups with acceleration structure items

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_NV_ray_tracing`

## New Object Types

- `VkAccelerationStructureNV`

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_GEOMETRY_NV`
  - `VK_STRUCTURE_TYPE_GEOMETRY_TRIANGLES_NV`
  - `VK_STRUCTURE_TYPE_GEOMETRY_AABB_NV`
  - `VK_STRUCTURE_TYPE_BIND_ACCELERATION_STRUCTURE_MEMORY_INFO_NV`
  - `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_NV`
  - `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_INFO_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PROPERTIES_NV`
  - `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_INFO_NV`
  - `VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_NV`

- Extending [VkShaderStageFlagBits](#):
  - `VK_SHADER_STAGE_RAYGEN_BIT_NV`
  - `VK_SHADER_STAGE_ANY_HIT_BIT_NV`
  - `VK_SHADER_STAGE_CLOSEST_HIT_BIT_NV`
  - `VK_SHADER_STAGE_MISS_BIT_NV`
  - `VK_SHADER_STAGE_INTERSECTION_BIT_NV`
  - `VK_SHADER_STAGE_CALLABLE_BIT_NV`
- Extending [VkPipelineStageFlagBits](#):
  - `VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV`
  - `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV`
- Extending [VkBufferUsageFlagBits](#):
  - `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV`
- Extending [VkPipelineBindPoint](#):
  - `VK_PIPELINE_BIND_POINT_RAY_TRACING_NV`
- Extending [VkDescriptorType](#):
  - `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV`
- Extending [VkAccessFlagBits](#):
  - `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NV`
  - `VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NV`
- Extending [VkQueryType](#):
  - `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV`
- Extending [VkPipelineCreateFlagBits](#):
  - `VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV`
- Extending [VkIndexType](#):
  - `VK_INDEX_TYPE_NONE_NV`

## New Enums

- [VkGeometryFlagBitsNV](#)
- [VkGeometryInstanceFlagBitsNV](#)
- [VkBuildAccelerationStructureFlagBitsNV](#)
- [VkCopyAccelerationStructureModeNV](#)
- [VkGeometryTypeNV](#)
- [VkRayTracingShaderGroupTypeNV](#)
- [VkAccelerationStructureMemoryRequirementsTypeNV](#)
- [VkAccelerationStructureTypeNV](#)

## New Structures

- [VkRayTracingPipelineCreateInfoNV](#)

- [VkGeometryTrianglesNV](#)
- [VkGeometryAABB NV](#)
- [VkGeometryDataNV](#)
- [VkGeometryNV](#)
- [VkAccelerationStructureCreateInfoNV](#)
- [VkBindAccelerationStructureMemoryInfoNV](#)
- [VkWriteDescriptorSetAccelerationStructureNV](#)
- [VkAccelerationStructureMemoryRequirementsInfoNV](#)
- [VkPhysicalDeviceRayTracingPropertiesNV](#)
- [VkRayTracingShaderGroupCreateInfoNV](#)
- [VkAccelerationStructureInfoNV](#)

## New Functions

- [vkCreateAccelerationStructureNV](#)
- [vkDestroyAccelerationStructureNV](#)
- [vkGetAccelerationStructureMemoryRequirementsNV](#)
- [vkBindAccelerationStructureMemoryNV](#)
- [vkCmdBuildAccelerationStructureNV](#)
- [vkCmdCopyAccelerationStructureNV](#)
- [vkCmdTraceRaysNV](#)
- [vkCreateRayTracingPipelinesNV](#)
- [vkGetRayTracingShaderGroupHandlesNV](#)
- [vkGetAccelerationStructureHandleNV](#)
- [vkCmdWriteAccelerationStructuresPropertiesNV](#)
- [vkCompileDeferredNV](#)

## New or Modified Built-In Variables

- [LaunchID NV](#)
- [LaunchSize NV](#)
- [WorldRayOrigin NV](#)
- [WorldRayDirection NV](#)
- [ObjectRayOrigin NV](#)
- [ObjectRayDirection NV](#)
- [RayTmin NV](#)
- [RayTmax NV](#)
- [InstanceCustomIndex NV](#)
- [InstanceId](#)

- `ObjectToWorldNV`
- `WorldToObjectNV`
- `HitTNV`
- `HitKindNV`
- `IncomingRayFlagsNV`
- (modified) `PrimitiveId`

## New SPIR-V Capabilities

- `RayTracingNV`

## Issues

- 1) Are there issues?

**RESOLVED:** Yes.

## Sample Code

Example ray generation GLSL shader

```
#version 450 core
#extension GL_NV_ray_tracing : require
layout(set = 0, binding = 0, rgba8) uniform image2D image;
layout(set = 0, binding = 1) uniform accelerationStructureNV as;
layout(location = 0) rayPayloadNV float payload;

void main()
{
    vec4 col = vec4(0, 0, 0, 1);

    vec3 origin = vec3(float(gl_LaunchIDNV.x)/float(gl_LaunchSizeNV.x), float
(gl_LaunchIDNV.y)/float(gl_LaunchSizeNV.y), 1.0);
    vec3 dir = vec3(0.0, 0.0, -1.0);

    traceNV(as, 0, 0xff, 0, 1, 0, origin, 0.0, dir, 1000.0, 0);

    col.y = payload;

    imageStore(image, ivec2(gl_LaunchIDNV.xy), col);
}
```

## Version History

- Revision 1, 2018-09-11 (Robert Stepinski, Nuno Subtil, Eric Werness)
  - Internal revisions
- Revision 2, 2018-10-19 (Eric Werness)

- rename to VK\_NV\_ray\_tracing, add support for callables.
  - too many updates to list
- Revision 3, 2018-11-20 (Daniel Koch)
    - update to use InstanceId instead of InstanceIndex as implemented.

## **VK\_NV\_representative\_fragment\_test**

### **Name String**

`VK_NV_representative_fragment_test`

### **Extension Type**

Device extension

### **Registered Extension Number**

167

### **Revision**

2

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Contact**

- Kedarnath Thangudu [Okthangudu](#)

### **Last Modified Date**

2018-09-13

### **Contributors**

- Kedarnath Thangudu, NVIDIA
- Christoph Kubisch, NVIDIA
- Pierre Boudier, NVIDIA
- Pat Brown, NVIDIA
- Jeff Bolz, NVIDIA
- Eric Werness, NVIDIA

This extension provides a new representative fragment test that allows implementations to reduce the amount of rasterization and fragment processing work performed for each point, line, or triangle primitive. For any primitive that produces one or more fragments that pass all other early fragment tests, the implementation is permitted to choose one or more “representative” fragments for processing and discard all other fragments. For draw calls rendering multiple points, lines, or triangles arranged in lists, strips, or fans, the representative fragment test is performed independently for each of those primitives.

This extension is useful for applications that use an early render pass to determine the full set of primitives that would be visible in the final scene. In this render pass, such applications would set

up a fragment shader that enables early fragment tests and writes to an image or shader storage buffer to record the ID of the primitive that generated the fragment. Without this extension, the shader would record the ID separately for each visible fragment of each primitive. With this extension, fewer stores will be performed, particularly for large primitives.

The representative fragment test has no effect if early fragment tests are not enabled via the fragment shader. The set of fragments discarded by the representative fragment test is implementation-dependent and may vary from frame to frame. In some cases, the representative fragment test may not discard any fragments for a given primitive.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#)
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_REPRESENTATIVE_FRAGMENT_TEST_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_PIPELINE_REPRESENTATIVE_FRAGMENT_TEST_STATE_CREATE_INFO_NV`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV](#)
- [VkPipelineRepresentativeFragmentTestStateCreateInfoNV](#)

## New Functions

None.

## Issues

(1) Is the representative fragment test guaranteed to have any effect?

**RESOLVED:** No. As specified, we only guarantee that each primitive with at least one fragment that passes prior tests will have one fragment passing the representative fragment tests. We don't guarantee that any particular fragment will fail the test.

In the initial implementation of this extension, the representative fragment test is treated as an optimization that may be completely disabled for some pipeline states. This feature was designed for a use case where the fragment shader records information on individual primitives using shader storage buffers or storage images, with no writes to color or depth buffers.

(2) Will the set of fragments that pass the representative fragment test be repeatable if you draw the same scene over and over again?

**RESOLVED:** No. The set of fragments that pass the representative fragment test is implementation-dependent and may vary due to the timing of operations performed by the GPU.

(3) What happens if you enable the representative fragment test with writes to color and/or depth render targets enabled?

**RESOLVED:** If writes to the color or depth buffer are enabled, they will be performed for any fragments that survive the relevant tests. Any fragments that fail the representative fragment test will not update color buffers. For the use cases intended for this feature, we don't expect color or depth writes to be enabled.

(4) How do derivatives and automatic texture level of detail computations work with the representative fragment test enabled?

**RESOLVED:** If a fragment shader uses derivative functions or texture lookups using automatic level of detail computation, derivatives will be computed identically whether or not the representative fragment test is enabled. For the use cases intended for this feature, we don't expect the use of derivatives in the fragment shader.

## Version History

- Revision 2, 2018-09-13 (pbrown)
  - Add issues.
- Revision 1, 2018-08-22 (Kedarnath Thangudu)
  - Internal Revisions

## VK\_NV\_sample\_mask\_override\_coverage

### Name String

`VK_NV_sample_mask_override_coverage`

### Extension Type

Device extension

### Registered Extension Number

95

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Piers Daniell [Opdaniell-nv](#)

### Last Modified Date

2016-12-08

## IP Status

No known IP claims.

## Interactions and External Dependencies

- This extension requires the [SPV\\_NV\\_sample\\_mask\\_override\\_coverage](#) SPIR-V extension.
- This extension requires the [GL\\_NV\\_sample\\_mask\\_override\\_coverage](#) extension for GLSL source languages.

## Contributors

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

This extension adds support for the following SPIR-V extension in Vulkan:

- [SPV\\_NV\\_sample\\_mask\\_override\\_coverage](#)

The extension provides access to the `OverrideCoverageNV` decoration under the `SampleMaskOverrideCoverageNV` capability. Adding this decoration to a variable with the `SampleMask` builtin decoration allows the shader to modify the coverage mask and affect which samples are used to process the fragment.

When using GLSL source-based shader languages, the `override_coverage` layout qualifier from `GL_NV_sample_mask_override_coverage` maps to the `OverrideCoverageNV` decoration. To use the `override_coverage` layout qualifier in GLSL the [GL\\_NV\\_sample\\_mask\\_override\\_coverage](#) extension must be enabled. Behavior is described in the [GL\\_NV\\_sample\\_mask\\_override\\_coverage](#) extension spec.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New Built-In Variables

None.

## New Variable Decoration

- [OverrideCoverageNV](#) in [SampleMask](#)

## New SPIR-V Capabilities

- [SampleMaskOverrideCoverageNV](#)

## Issues

None.

## Version History

- Revision 1, 2016-12-08 (Piers Daniell)
  - Internal revisions

## **VK\_NV\_scissor\_exclusive**

### Name String

`VK_NV_scissor_exclusive`

### Extension Type

Device extension

### Registered Extension Number

206

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Contact

- Pat Brown [Onvpbrown](#)

### Last Modified Date

2018-07-31

### IP Status

No known IP claims.

### Interactions and External Dependencies

None

### Contributors

- Pat Brown, NVIDIA
- Jeff Bolz, NVIDIA
- Piers Daniell, NVIDIA
- Daniel Koch, NVIDIA

This extension adds support for an exclusive scissor test to Vulkan. The exclusive scissor test behaves like the scissor test, except that the exclusive scissor test fails for pixels inside the corresponding rectangle and passes for pixels outside the rectangle. If the same rectangle is used for both the scissor and exclusive scissor tests, the exclusive scissor test will pass if and only if the scissor test fails.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#)
  - `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_EXCLUSIVE_SCISSOR_STATE_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXCLUSIVE_SCISSOR_FEATURES_NV`
- Extending [VkDynamicState](#)
  - `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceExclusiveScissorFeaturesNV](#)
- [VkPipelineViewportExclusiveScissorStateCreateInfoNV](#)

## New Functions

- [vkCmdSetExclusiveScissorNV](#)

## New Built-In Variables

None.

## New SPIR-V Capabilities

None.

## Issues

1) For the scissor test, the viewport state must be created with a matching number of scissor and viewport rectangles. Should we have the same requirement for exclusive scissors?

**RESOLVED:** For exclusive scissors, we relax this requirement and allow an exclusive scissor rectangle count that is either zero or equal to the number of viewport rectangles. If you pass in an exclusive scissor count of zero, the exclusive scissor test is treated as disabled.

## Version History

- Revision 1, 2018-07-31 (Pat Brown)
  - Internal revisions

## VK\_NV\_shader\_image\_footprint

### Name String

`VK_NV_shader_image_footprint`

### Extension Type

Device extension

### Registered Extension Number

205

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Contact

- Pat Brown [Onvpbrown](#)

### Last Modified Date

2018-09-13

### IP Status

No known IP claims.

### Contributors

- Pat Brown, NVIDIA
- Chris Lentini, NVIDIA
- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

This extension adds Vulkan support for the `SPV_NV_shader_image_footprint` SPIR-V extension. That SPIR-V extension provides a new instruction `OpImageSampleFootprintNV` allowing shaders to determine the set of texels that would be accessed by an equivalent filtered texture lookup.

Instead of returning a filtered texture value, the instruction returns a structure that can be interpreted by shader code to determine the footprint of a filtered texture lookup. This structure includes integer values that identify a small neighborhood of texels in the image being accessed and a bitfield that indicates which texels in that neighborhood would be used. The structure also includes a bitfield where each bit identifies whether any texel in a small aligned block of texels would be fetched by the texture lookup. The size of each block is specified by an access *granularity* provided by the shader. The minimum granularity supported by this extension is 2x2 (for 2D textures) and 2x2x2 (for 3D textures); the maximum granularity is 256x256 (for 2D textures) or 64x32x32 (for 3D textures). Each footprint query returns the footprint from a single texture level. When using minification filters that combine accesses from multiple mipmap levels, shaders must perform separate queries for the two levels accessed (“fine” and “coarse”). The footprint query also returns a flag indicating if the texture lookup would access texels from only one mipmap level or from two neighboring levels.

This extension should be useful for multi-pass rendering operations that do an initial expensive rendering pass to produce a first image that is then used as a texture for a second pass. If the second pass ends up accessing only portions of the first image (e.g., due to visibility), the work spent rendering the non-accessed portion of the first image was wasted. With this feature, an application can limit this waste using an initial pass over the geometry in the second image that performs a footprint query for each visible pixel to determine the set of pixels that it needs from the first image. This pass would accumulate an aggregate footprint of all visible pixels into a separate “footprint image” using shader atomics. Then, when rendering the first image, the application can kill all shading work for pixels not in this aggregate footprint.

This extension has a number of limitations. The `OpImageSampleFootprintNV` instruction only supports for two- and three-dimensional textures. Footprint evaluation only supports the `CLAMP_TO_EDGE` wrap mode; results are undefined for all other wrap modes. Only a limited set of granularity values and that set does not support separate coverage information for each texel in the original image.

When using SPIR-V generated from the OpenGL Shading Language, the new instruction will be generated from code using the new `textureFootprint*NV` built-in functions from the `GL_NV_shader_texture_footprint` shading language extension.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_FOOTPRINT_FEATURES_NV`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceShaderImageFootprintFeaturesNV](#)

## New Functions

None.

## New SPIR-V Capability

- [ImageFootprintNV](#)

## Issues

(1) The footprint returned by the SPIR-V instruction is a structure that includes an anchor, an offset, and a mask that represents a 8x8 or 4x4x4 neighborhood of texel groups. But the bits of the mask are not stored in simple pitch order. Why is the footprint built this way?

**RESOLVED:** We expect that applications using this feature will want to use a fixed granularity and accumulate coverage information from the returned footprints into an aggregate “footprint image” that tracks the portions of an image that would be needed by regular texture filtering. If an application is using a two-dimensional image with 4x4 pixel granularity, we expect that the footprint image will use 64-bit texels where each bit in an 8x8 array of bits corresponds to coverage for a 4x4 block in the original image. Texel (0,0) in the footprint image would correspond to texels (0,0) through (31,31) in the original image.

In the usual case, the footprint for a single access will fully contained in a 32x32 aligned region of the original texture, which corresponds to a single 64-bit texel in the footprint image. In that case, the implementation will return an anchor coordinate pointing at the single footprint image texel, an offset vector of (0,0), and a mask whose bits are aligned with the bits in the footprint texel. For this case, the shader can simply atomically OR the mask bits into the contents of the footprint texel to accumulate footprint coverage.

In the worst case, the footprint for a single access spans multiple 32x32 aligned regions and may require updates to four separate footprint image texels. In this case, the implementation will return an anchor coordinate pointing at the lower right footprint image texel and an offset will identify how many “columns” and “rows” of the returned 8x8 mask correspond to footprint texels to the left and above the anchor texel. If the anchor is (2,3), the 64 bits of the returned mask are arranged spatially as follows, where each 4x4 block is assigned a bit number that matches its bit number in the footprint image texels:

To accumulate coverage for each of the four footprint image texels, a shader can AND the returned mask with simple masks derived from the x and y offset values and then atomically OR the updated mask bits into the contents of the corresponding footprint texel.

```

    uint64_t returnedMask = (uint64_t(footprint.mask.x) | (uint64_t(footprint.mask.y)
<< 32));
    uint64_t rightMask    = ((0xFF >> footprint.offset.x) * 0x0101010101010101UL);
    uint64_t bottomMask   = 0xFFFFFFFFFFFFFFFUL >> (8 * footprint.offset.y);
    uint64_t bottomRight  = returnedMask & bottomMask & rightMask;
    uint64_t bottomLeft   = returnedMask & bottomMask & (~rightMask);
    uint64_t topRight     = returnedMask & (~bottomMask) & rightMask;
    uint64_t topLeft      = returnedMask & (~bottomMask) & (~rightMask);

```

(2) What should an application do to ensure maximum performance when accumulating footprints into an aggregate footprint image?

**RESOLVED:** We expect that the most common usage of this feature will be to accumulate aggregate footprint coverage, as described in the previous issue. Even if you ignore the anisotropic filtering case where the implementation may return a granularity larger than that requested by the caller, each shader invocation will need to use atomic functions to update up to four footprint image texels for each level of detail accessed. Having each active shader invocation perform multiple atomic operations can be expensive, particularly when neighboring invocations will want to update the same footprint image texels.

Techniques can be used to reduce the number of atomic operations performed when accumulating coverage include:

- Have logic that detects returned footprints where all components of the returned offset vector

are zero. In that case, the mask returned by the footprint function is guaranteed to be aligned with the footprint image texels and affects only a single footprint image texel.

- Have fragment shaders communicate using built-in functions from the `VK_NV_shader_subgroup_partitioned` extension or other shader subgroup extensions. If you have multiple invocations in a subgroup that need to update the same texel (x,y) in the footprint image, compute an aggregate footprint mask across all invocations in the subgroup updating that texel and have a single invocation perform an atomic operation using that aggregate mask.
- When the returned footprint spans multiple texels in the footprint image, each invocation need to perform four atomic operations. In the previous issue, we had an example that computed separate masks for “topLeft”, “topRight”, “bottomLeft”, and “bottomRight”. When the invocations in a subgroup have good locality, it might be the case the “top left” for some invocations might refer to footprint image texel (10,10), while neighbors might have their “top left” texels at (11,10), (10,11), and (11,11). If you compute separate masks for even/odd x and y values instead of left/right or top/bottom, the “odd/odd” mask for all invocations in the subgroup hold coverage for footprint image texel (11,11), which can be updated by a single atomic operation for the entire subgroup.

## Examples

TBD

## Version History

- Revision 2, 2018-09-13 (Pat Brown)
  - Add issue (2) with performance tips.
- Revision 1, 2018-08-12 (Pat Brown)
  - Initial draft

## `VK_NV_shader_sm_builtins`

### Name String

`VK_NV_shader_sm_builtins`

### Extension Type

Device extension

### Registered Extension Number

155

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.1

### Contact

- Daniel Koch [Odgkoch](#)

## Last Modified Date

2019-05-28

## Interactions and External Dependencies

- This extension requires `SPV_NV_shader_sm_builtins`.
- This extension enables `GL_NV_shader_sm_builtins` for GLSL source languages.

## Contributors

- Jeff Bolz, NVIDIA
- Eric Werness, NVIDIA

## Description

This extension provides the ability to determine device-specific properties on NVIDIA GPUs. It provides the number of streaming multiprocessors (SMs), the maximum number of warps (subgroups) that can run on an SM, and shader builtins to enable invocations to identify which SM and warp a shader invocation is executing on.

This extension enables support for the SPIR-V `ShaderSMBuiltinsNV` capability.

These properties and built-ins **should** typically only be used for debugging purposes.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_PROPERTIES_NV`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV`
- `VkPhysicalDeviceShaderSMBuiltinsPropertiesNV`

## New Functions

None.

## New or Modified Built-In Variables

- `WarpPerSMNV`
- `SMCountNV`
- `WarpIDNV`
- `SMIDNV`

## New SPIR-V Capabilities

- `ShaderSMBuiltinsNV`

## Issues

1. What should we call this extension?

RESOLVED: Using NV\_shader\_sm\_builtin. Other options considered included:

- NV\_shader\_smid - but SMID is really easy to typo/confuse as SIMD.
- NV\_shader\_sm\_info - but **Info** is typically reserved for input structures

## Version History

- Revision 1, 2019-05-28 (Daniel Koch)
  - Internal revisions

## VK\_NV\_shader\_subgroup\_partitioned

### Name String

`VK_NV_shader_subgroup_partitioned`

### Extension Type

Device extension

### Registered Extension Number

199

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.1

### Contact

- Jeff Bolz [@jeffbolz\\_nv](#)

### Last Modified Date

2018-03-17

## Contributors

- Jeff Bolz, NVIDIA

This extension enables support for a new class of subgroup operations via the `GL_NV_shader_subgroup_partitioned` GLSL extension and `SPV_NV_shader_subgroup_partitioned` SPIR-V extension. Support for these new operations is advertised via the `VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV` bit.

This extension requires Vulkan 1.1, for general subgroup support.

## New Object Types

None.

## New Enum Constants

- Extending `VkSubgroupFeatureFlagBits`:
  - `VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV`

## New Enums

None.

## New Structures

None.

## New Functions

None.

## Issues

None.

## Version History

- Revision 1, 2018-03-17 (Jeff Bolz)
  - Internal revisions

## `VK_NV_shading_rate_image`

### Name String

`VK_NV_shading_rate_image`

### Extension Type

Device extension

## Registered Extension Number

165

## Revision

3

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

## Contact

- Pat Brown [Onvpbrown](#)

## Last Modified Date

2019-07-18

## Contributors

- Pat Brown, NVIDIA
- Carsten Rohde, NVIDIA
- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA
- Mathias Schott, NVIDIA
- Matthew Netsch, Qualcomm Technologies, Inc.

This extension allows applications to use a variable shading rate when processing fragments of rasterized primitives. By default, Vulkan will spawn one fragment shader for each pixel covered by a primitive. In this extension, applications can bind a *shading rate image* that can be used to vary the number of fragment shader invocations across the framebuffer. Some portions of the screen may be configured to spawn up to 16 fragment shaders for each pixel, while other portions may use a single fragment shader invocation for a 4x4 block of pixels. This can be useful for use cases like eye tracking, where the portion of the framebuffer that the user is looking at directly can be processed at high frequency, while distant corners of the image can be processed at lower frequency. Each texel in the shading rate image represents a fixed-size rectangle in the framebuffer, covering 16x16 pixels in the initial implementation of this extension. When rasterizing a primitive covering one of these rectangles, the Vulkan implementation reads a texel in the bound shading rate image and looks up the fetched value in a palette to determine a base shading rate.

In addition to the API support controlling rasterization, this extension also adds Vulkan support for the `SPV_NV_shading_rate` extension to SPIR-V. That extension provides two fragment shader variable decorations that allow fragment shaders to determine the shading rate used for processing the fragment:

- `FragmentSizeNV`, which indicates the width and height of the set of pixels processed by the fragment shader.
- `InvocationsPerPixel`, which indicates the maximum number of fragment shader invocations that could be spawned for the pixel(s) covered by the fragment.

When using SPIR-V in conjunction with the OpenGL Shading Language (GLSL), the fragment shader capabilities are provided by the `GL_NV_shading_rate_image` language extension and correspond to the built-in variables `gl_FragmentSizeNV` and `gl_InvocationsPerPixelNV`, respectively.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SHADING_RATE_IMAGE_STATE_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_PROPERTIES_NV`
  - `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_COARSE_SAMPLE_ORDER_STATE_CREATE_INFO_NV`
- Extending `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV`
- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV`
  - `VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV`
- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_SHADING_RATE_IMAGE_READ_BIT_NV`
- Extending `VkImageUsageFlagBits`:
  - `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`
- Extending `VkPipelineStageFlagBits`
  - `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`

## New Enums

- `VkShadingRatePaletteEntryNV`, containing the following constants:
  - `VK_SHADING_RATE_PALETTE_ENTRY_NO_INVOCATIONS_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_16_INVOCATIONS_PER_PIXEL_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_8_INVOCATIONS_PER_PIXEL_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_4_INVOCATIONS_PER_PIXEL_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_2_INVOCATIONS_PER_PIXEL_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_PIXEL_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X1_PIXELS_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_1X2_PIXELS_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X2_PIXELS_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X2_PIXELS_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X4_PIXELS_NV`
  - `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X4_PIXELS_NV`
- `VkCoarseSampleOrderTypeNV`, containing the following constants:

- `VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV`
- `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV`
- `VK_COARSE_SAMPLE_ORDER_TYPE_PIXEL_MAJOR_NV`
- `VK_COARSE_SAMPLE_ORDER_TYPE_SAMPLE_MAJOR_NV`

## New Structures

- [VkShadingRatePaletteNV](#)
- [VkPipelineViewportShadingRateImageStateCreateInfoNV](#)
- [VkPhysicalDeviceShadingRateImageFeaturesNV](#)
- [VkPhysicalDeviceShadingRateImagePropertiesNV](#)
- [VkCoarseSampleLocationNV](#)
- [VkCoarseSampleOrderCustomNV](#)
- [VkPipelineViewportCoarseSampleOrderStateCreateInfoNV](#)

## New Functions

- [vkCmdBindShadingRateImageNV](#)
- [vkCmdSetViewportShadingRatePaletteNV](#)
- [vkCmdSetCoarseSampleOrderNV](#)

## Issues

(1) When using shading rates specifying “coarse” fragments covering multiple pixels, we will generate a combined coverage mask that combines the coverage masks of all pixels covered by the fragment. By default, these masks are combined in an implementation-dependent order. Should we provide a mechanism allowing applications to query or specify an exact order?

**RESOLVED:** Yes, this feature is useful for cases where most of the fragment shader can be evaluated once for an entire coarse fragment, but where some per-pixel computations are also required. For example, a per-pixel alpha test may want to kill all the samples for some pixels in a coarse fragment. This sort of test can be implemented using an output sample mask, but such a shader would need to know which bit in the mask corresponds to each sample in the coarse fragment. We are including a mechanism to allow applications to specify the orders of coverage samples for each shading rate and sample count, either as static pipeline state or dynamically via a command buffer. This portion of the extension has its own feature bit.

We will not be providing a query to determine the implementation-dependent default ordering. The thinking here is that if an application cares enough about the coarse fragment sample ordering to perform such a query, it could instead just set its own order, also using custom per-pixel sample locations if required.

(2) For the pipeline stage `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`, should we specify a precise location in the pipeline the shading rate image is accessed (after geometry shading, but before the early fragment tests) or leave it under-specified in case there are other implementations that access

the image in a different pipeline location?

**RESOLVED** We are specifying the pipeline stage to be between the final stage used for vertex processing (`VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`) and before the first stage used for fragment processing (`VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`), which seems to be the natural place to access the shading rate image.

(3) How do centroid-sampled variables work with fragments larger than one pixel?

**RESOLVED** For single-pixel fragments, fragment shader inputs decorated with `Centroid` are sampled at an implementation-dependent location in the intersection of the area of the primitive being rasterized and the area of the pixel that corresponds to the fragment. With multi-pixel fragments, we follow a similar pattern, using the intersection of the primitive and the `set` of pixels corresponding to the fragment.

One important thing to keep in mind when using such “coarse” shading rates is that fragment attributes are sampled at the center of the fragment by default, regardless of the set of pixels/samples covered by the fragment. For fragments with a size of 4x4 pixels, this center location will be more than two pixels ( $1.5 * \sqrt{2}$ ) away from the center of the pixels at the corners of the fragment. When rendering a primitive that covers only a small part of a coarse fragment, sampling a color outside the primitive can produce overly bright or dark color values if the color values have a large gradient. To deal with this, an application can use centroid sampling on attributes where “extrapolation” artifacts can lead to overly bright or dark pixels. Note that this same problem also exists for multisampling with single-pixel fragments, but is less severe because it only affects certain samples of a pixel and such bright/dark samples may be averaged with other samples that don’t have a similar problem.

## Version History

- Revision 3, 2019-07-18 (Mathias Schott)
  - Fully list extension interfaces in this appendix.
- Revision 2, 2018-09-13 (Pat Brown)
  - Miscellaneous edits preparing the specification for publication.
- Revision 1, 2018-08-08 (Pat Brown)
  - Internal revisions

## `VK_NV_viewport_array2`

### Name String

`VK_NV_viewport_array2`

### Extension Type

Device extension

### Registered Extension Number

97

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Contact

- Daniel Koch [dgkoch](#)

## Last Modified Date

2017-02-15

## Interactions and External Dependencies

- This extension requires the `SPV_NV_viewport_array2` SPIR-V extension.
- This extension requires the `GL_NV_viewport_array2` extension for GLSL source languages.
- This extension requires the `geometryShader` and `multiViewport` features.
- This extension interacts with the `tessellationShader` feature.

## Contributors

- Piers Daniell, NVIDIA
- Jeff Bolz, NVIDIA

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_NV_viewport_array2`

which allows a single primitive to be broadcast to multiple viewports and/or multiple layers. A new shader built-in output `ViewportMaskNV` is provided, which allows a single primitive to be output to multiple viewports simultaneously. Also, a new SPIR-V decoration is added to control whether the effective viewport index is added into the variable decorated with the `Layer` built-in decoration. These capabilities allow a single primitive to be output to multiple layers simultaneously.

This extension allows variables decorated with the `Layer` and `ViewportIndex` built-ins to be exported from vertex or tessellation shaders, using the `ShaderViewportIndexLayerNV` capability.

This extension adds a new `ViewportMaskNV` built-in decoration that is available for output variables in vertex, tessellation evaluation, and geometry shaders, and a new `ViewportRelativeNV` decoration that can be added on variables decorated with `Layer` when using the `ShaderViewportMaskNV` capability.

When using GLSL source-based shading languages, the `gl_ViewportMask[]` built-in output variable and `viewport_relative` layout qualifier from `GL_NV_viewport_array2` map to the `ViewportMaskNV` and `ViewportRelativeNV` decorations, respectively. Behaviour is described in the `GL_NV_viewport_array2` extension specification.

*Note*



The `ShaderViewportIndexLayerNV` capability is equivalent to the `ShaderViewportIndexLayerEXT` capability added by `VK_EXT_shader_viewport_index_layer`.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New or Modified Built-In Variables

- (modified) `Layer`
- (modified) `ViewportIndex`
- `ViewportMaskNV`

## New Variable Decoration

- `ViewportRelativeNV` in `Layer`

## New SPIR-V Capabilities

- `ShaderViewportIndexLayerNV`
- `ShaderViewportMaskNV`

## Issues

None yet!

## Version History

- Revision 1, 2017-02-15 (Daniel Koch)

- Internal revisions

## VK\_NV\_viewport\_swizzle

### Name String

`VK_NV_viewport_swizzle`

### Extension Type

Device extension

### Registered Extension Number

99

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Contact

- Piers Daniell [Opdaniell-nv](#)

### Last Modified Date

2016-12-22

### Interactions and External Dependencies

- This extension requires `multiViewport` and `geometryShader` features to be useful.

### Contributors

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

This extension provides a new per-viewport swizzle that can modify the position of primitives sent to each viewport. New viewport swizzle state is added for each viewport, and a new position vector is computed for each vertex by selecting from and optionally negating any of the four components of the original position vector.

This new viewport swizzle is useful for a number of algorithms, including single-pass cubemap rendering (broadcasting a primitive to multiple faces and reorienting the vertex position for each face) and voxel rasterization. The per-viewport component remapping and negation provided by the swizzle allows application code to re-orient three-dimensional geometry with a view along any of the X, Y, or Z axes. If a perspective projection and depth buffering is required, 1/W buffering should be used, as described in the single-pass cubemap rendering example in the “Issues” section below.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SWIZZLE_STATE_CREATE_INFO_NV`

## New Enums

- [VkViewportCoordinateSwizzleNV](#)
- [VkPipelineViewportSwizzleStateCreateFlagsNV](#)

## New Structures

- [VkViewportSwizzleNV](#)
- [VkPipelineViewportSwizzleStateCreateInfoNV](#)

## New Functions

None.

## Issues

1) Where does viewport swizzling occur in the pipeline?

**RESOLVED:** Despite being associated with the viewport, viewport swizzling must happen prior to the viewport transform. In particular, it needs to be performed before clipping and perspective division.

The viewport mask expansion ([VK\\_NV\\_viewport\\_array2](#)) and the viewport swizzle could potentially be performed before or after transform feedback, but feeding back several viewports worth of primitives with different swizzles doesn't seem particularly useful. This specification applies the viewport mask and swizzle after transform feedback, and makes primitive queries only count each primitive once.

2) Any interesting examples of how this extension, [VK\\_NV\\_viewport\\_array2](#), and [VK\\_NV\\_geometry\\_shader\\_passthrough](#) can be used together in practice?

**RESOLVED:** One interesting use case for this extension is for single-pass rendering to a cubemap. In this example, the application would attach a cubemap texture to a layered FBO where the six cube faces are treated as layers. Vertices are sent through the vertex shader without applying a projection matrix, where the `gl_Position` output is (x,y,z,1) and the center of the cubemap is at (0,0,0). With unextended Vulkan, one could have a conventional instanced geometry shader that looks something like the following:

```

layout(invocations = 6) in;      // separate invocation per face
layout(triangles) in;
layout(triangle_strip) out;
layout(max_vertices = 3) out;

in Inputs {
vec2 texcoord;
vec3 normal;
vec4 baseColor;
} v[];

out Outputs {
vec2 texcoord;
vec3 normal;
vec4 baseColor;
};

void main()
{
int face = gl_InvocationID; // which face am I?

// Project gl_Position for each vertex onto the cube map face.
vec4 positions[3];
for (int i = 0; i < 3; i++) {
    positions[i] = rotate(gl_in[i].gl_Position, face);
}

// If the primitive doesn't project onto this face, we're done.
if (shouldCull(positions)) {
    return;
}

// Otherwise, emit a copy of the input primitive to the
// appropriate face (using gl_Layer).
for (int i = 0; i < 3; i++) {
    gl_Layer = face;
    gl_Position = positions[i];
    texcoord = v[i].texcoord;
    normal = v[i].normal;
    baseColor = v[i].baseColor;
    EmitVertex();
}
}

```

With passthrough geometry shaders, this can be done using a much simpler shader:

```

layout(triangles) in;
layout(passthrough) in Inputs {
    vec2 texcoord;
    vec3 normal;
    vec4 baseColor;
}
layout(passthrough) in gl_PerVertex {
    vec4 gl_Position;
} gl_in[];
layout(viewport_relative) out int gl_Layer;

void main()
{
    // Figure out which faces the primitive projects onto and
    // generate a corresponding viewport mask.
    uint mask = 0;
    for (int i = 0; i < 6; i++) {
        if (!shouldCull(face)) {
            mask |= 1U << i;
        }
    }
    gl_ViewportMask = mask;
    gl_Layer = 0;
}

```

The application code is set up so that each of the six cube faces has a separate viewport (numbered 0 to 5). Each face also has a separate swizzle, programmed via the `VkPipelineViewportSwizzleStateCreateInfoNV` pipeline state. The viewport swizzle feature performs the coordinate transformation handled by the `rotate()` function in the original shader. The `viewport_relative` layout qualifier says that the viewport number (0 to 5) is added to the base `gl_Layer` value of 0 to determine which layer (cube face) the primitive should be sent to.

Note that the use of the passed through input `normal` in this example suggests that the fragment shader in this example would perform an operation like per-fragment lighting. The viewport swizzle would transform the position to be face-relative, but `normal` would remain in the original coordinate system. It seems likely that the fragment shader in either version of the example would want to perform lighting in the original coordinate system. It would likely do this by reconstructing the position of the fragment in the original coordinate system using `gl_FragCoord`, a constant or uniform holding the size of the cube face, and the input `gl_ViewportIndex` (or `gl_Layer`), which identifies the cube face. Since the value of `normal` is in the original coordinate system, it would not need to be modified as part of this coordinate transformation.

Note that while the `rotate()` operation in the regular geometry shader above could include an arbitrary post-rotation projection matrix, the viewport swizzle does not support arbitrary math. To get proper projection, 1/W buffering should be used. To do this:

1. Program the viewport swizzles to move the pre-projection W eye coordinate (typically 1.0) into the Z coordinate of the swizzle output and the eye coordinate component used for depth into the W coordinate. For example, the viewport corresponding to the +Z face might use a swizzle of

(+X, -Y, +W, +Z). The Z normalized device coordinate computed after swizzling would then be  $z'/w' = 1/Z_{\text{eye}}$ .

2. On NVIDIA implementations supporting floating-point depth buffers with values outside [0,1], prevent unwanted near plane clipping by enabling `depthClampEnable`. Ensure that the depth clamp doesn't mess up depth testing by programming the depth range to very large values, such as `minDepthBounds=-z`, `maxDepthBounds=+z`, where  $z = 2^{127}$ . It should be possible to use IEEE infinity encodings also (`0xFF800000` for `-INF`, `0x7F800000` for `+INF`). Even when near/far clipping is disabled, primitives extending behind the eye will still be clipped because one or more vertices will have a negative W coordinate and fail X/Y clipping tests.

On other implementations, scale X, Y, and Z eye coordinates so that vertices on the near plane have a post-swizzle W coordinate of 1.0. For example, if the near plane is at  $Z_{\text{eye}} = 1/256$ , scale X, Y, and Z by 256.

3. Adjust depth testing to reflect the fact that  $1/W$  values are large near the eye and small away from the eye. Clear the depth buffer to zero (infinitely far away) and use a depth test of `VK_COMPARE_OP_GREATER` instead of `VK_COMPARE_OP_LESS`.

## Version History

- Revision 1, 2016-12-22 (Piers Daniell)
  - Internal revisions

## List of Deprecated Extensions

- [VK\\_KHR\\_16bit\\_storage](#)
- [VK\\_KHR\\_bind\\_memory2](#)
- [VK\\_KHR\\_dedicated\\_allocation](#)
- [VK\\_KHR\\_descriptor\\_update\\_template](#)
- [VK\\_KHR\\_device\\_group](#)
- [VK\\_KHR\\_device\\_group\\_creation](#)
- [VK\\_KHR\\_external\\_fence](#)
- [VK\\_KHR\\_external\\_fence\\_capabilities](#)
- [VK\\_KHR\\_external\\_memory](#)
- [VK\\_KHR\\_external\\_memory\\_capabilities](#)
- [VK\\_KHR\\_external\\_semaphore](#)
- [VK\\_KHR\\_external\\_semaphore\\_capabilities](#)
- [VK\\_KHR\\_get\\_memory\\_requirements2](#)
- [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)
- [VK\\_KHR\\_maintenance1](#)
- [VK\\_KHR\\_maintenance2](#)

- [VK\\_KHR\\_maintenance3](#)
- [VK\\_KHR\\_multiview](#)
- [VK\\_KHR\\_relaxed\\_block\\_layout](#)
- [VK\\_KHR\\_sampler\\_ycbcr\\_conversion](#)
- [VK\\_KHR\\_shader\\_draw\\_parameters](#)
- [VK\\_KHR\\_storage\\_buffer\\_storage\\_class](#)
- [VK\\_KHR\\_variable\\_pointers](#)
- [VK\\_EXT\\_buffer\\_device\\_address](#)
- [VK\\_EXT\\_debug\\_marker](#)
- [VK\\_EXT\\_debug\\_report](#)
- [VK\\_EXT\\_validation\\_flags](#)
- [VK\\_AMD\\_draw\\_indirect\\_count](#)
- [VK\\_AMD\\_gpu\\_shader\\_half\\_float](#)
- [VK\\_AMD\\_gpu\\_shader\\_int16](#)
- [VK\\_AMD\\_negative\\_viewport\\_height](#)
- [VK\\_NV\\_dedicated\\_allocation](#)
- [VK\\_NV\\_external\\_memory](#)
- [VK\\_NV\\_external\\_memory\\_capabilities](#)
- [VK\\_NV\\_external\\_memory\\_win32](#)
- [VK\\_NV\\_glsl\\_shader](#)
- [VK\\_NV\\_win32\\_keyed\\_mutex](#)

## **VK\_KHR\_16bit\_storage**

### **Name String**

`VK_KHR_16bit_storage`

### **Extension Type**

Device extension

### **Registered Extension Number**

84

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_storage_buffer_storage_class`

### **Deprecation state**

- *Promoted* to [Vulkan 1.1](#)

### **Contact**

- Jan-Harald Fredriksen [janharaldfredriksen-arm](#)

### **Last Modified Date**

2017-09-05

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

- This extension requires [SPV\\_KHR\\_16bit\\_storage](#)
- Promoted to Vulkan 1.1 Core

### **Contributors**

- Alexander Galazin, ARM
- Jan-Harald Fredriksen, ARM
- Joerg Wagner, ARM
- Neil Henning, Codeplay
- Jeff Bolz, Nvidia
- Daniel Koch, Nvidia
- David Neto, Google
- John Kessenich, Google

The `VK_KHR_16bit_storage` extension allows use of 16-bit types in shader input and output interfaces, and push constant blocks. This extension introduces several new optional features which map to SPIR-V capabilities and allow access to 16-bit data in `Block`-decorated objects in the `Uniform` and the `StorageBuffer` storage classes, and objects in the `PushConstant` storage class. This extension allows 16-bit variables to be declared and used as user-defined shader inputs and outputs but does not change location assignment and component assignment rules.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES_KHR`

## New Structures

- `VkPhysicalDevice16BitStorageFeaturesKHR`

## New SPIR-V Capabilities

- `StorageBuffer16BitAccess`
- `UniformAndStorageBuffer16BitAccess`
- `StoragePushConstant16`
- `StorageInputOutput16`

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

## Version History

- Revision 1, 2017-03-23 (Alexander Galazin)
  - Initial draft

## `VK_KHR_bind_memory2`

### Name String

`VK_KHR_bind_memory2`

### Extension Type

Device extension

### Registered Extension Number

158

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Deprecation state

- *Promoted* to [Vulkan 1.1](#)

## Contact

- Tobias Hector [@tobski](#)

## Last Modified Date

2017-09-05

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jeff Bolz, NVIDIA
- Tobias Hector, Imagination Technologies

This extension provides versions of [vkBindBufferMemory](#) and [vkBindImageMemory](#) that allow multiple bindings to be performed at once, and are extensible.

This extension also introduces [VK\\_IMAGE\\_CREATE\\_ALIAS\\_BIT\\_KHR](#), which allows “identical” images that alias the same memory to interpret the contents consistently, even across image layout changes.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_BIND\\_BUFFER\\_MEMORY\\_INFO\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_BIND\\_IMAGE\\_MEMORY\\_INFO\\_KHR](#)
- Extending [VkImageCreateFlagBits](#):
  - [VK\\_IMAGE\\_CREATE\\_ALIAS\\_BIT\\_KHR](#)

## New Enums

None.

## New Structures

- [VkBindBufferMemoryInfoKHR](#)

- [VkBindImageMemoryInfoKHR](#)

## New Functions

- [vkBindBufferMemory2KHR](#)
- [vkBindImageMemory2KHR](#)

## New Built-In Variables

None.

## New SPIR-V Capabilities

None.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Version History

- Revision 1, 2017-05-19 (Tobias Hector)
  - Pulled bind memory functions into their own extension

## VK\_KHR\_dedicated\_allocation

### Name String

`VK_KHR_dedicated_allocation`

### Extension Type

Device extension

### Registered Extension Number

128

### Revision

3

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_memory\\_requirements2](#)

## Deprecation state

- Promoted to Vulkan 1.1

## Contact

- James Jones [@cubanismo](#)

## Last Modified Date

2017-09-05

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jeff Bolz, NVIDIA
- Jason Ekstrand, Intel

This extension enables resources to be bound to a dedicated allocation, rather than suballocated. For any particular resource, applications **can** query whether a dedicated allocation is recommended, in which case using a dedicated allocation **may** improve the performance of access to that resource. Normal device memory allocations must support multiple resources per allocation, memory aliasing and sparse binding, which could interfere with some optimizations. Applications should query the implementation for when a dedicated allocation **may** be beneficial by adding `VkMemoryDedicatedRequirementsKHR` to the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter to a call to `vkGetBufferMemoryRequirements2` or `vkGetImageMemoryRequirements2`. Certain external handle types and external images or buffers **may** also depend on dedicated allocations on implementations that associate image or buffer metadata with OS-level memory objects.

This extension adds a two small structures to memory requirements querying and memory allocation: a new structure that flags whether an image/buffer should have a dedicated allocation, and a structure indicating the image or buffer that an allocation will be bound to.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS_KHR`
  - `VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO_KHR`

## New Enums

None.

## New Structures

- [VkMemoryDedicatedRequirementsKHR](#)
- [VkMemoryDedicatedAllocateInfoKHR](#)

## New Functions

None.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Examples

```
// Create an image with a dedicated allocation based on the
// implementation's preference

VkImageCreateInfo imageCreateInfo =
{
    // Image creation parameters
};

VkImage image;
VkResult result = vkCreateImage(
    device,
    &imageCreateInfo,
    NULL,                                // pAllocator
    &image);

VkMemoryDedicatedRequirementsKHR dedicatedRequirements =
{
    VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS_KHR,
    NULL,                                     // pNext
};

VkMemoryRequirements2 memoryRequirements =
{
    VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2,
    &dedicatedRequirements,                  // pNext
};

const VkImageMemoryRequirementsInfo2 imageRequirementsInfo =
{
```

```

VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2,
NULL, // pNext
image
};

vkGetImageMemoryRequirements2(
    device,
    &imageRequirementsInfo,
    &memoryRequirements);

if (dedicatedRequirements.prefersDedicatedAllocation) {
    // Allocate memory with VkMemoryDedicatedAllocateInfoKHR::image
    // pointing to the image we are allocating the memory for

    VkMemoryDedicatedAllocateInfoKHR dedicatedInfo =
    {
        VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO_KHR, // sType
        NULL, // pNext
        image, // image
        VK_NULL_HANDLE, // buffer
    };

    VkMemoryAllocateInfo memoryAllocateInfo =
    {
        VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO, // sType
        &dedicatedInfo, // pNext
        memoryRequirements.size, // allocationSize
        FindMemoryTypeIndex(memoryRequirements.memoryTypeBits), // memoryTypeIndex
    };

    VkDeviceMemory memory;
    vkAllocateMemory(
        device,
        &memoryAllocateInfo,
        NULL, // pAllocator
        &memory);
}

// Bind the image to the memory

vkBindImageMemory(
    device,
    image,
    memory,
    0);
} else {
    // Take the normal memory sub-allocation path
}

```

## Version History

- Revision 1, 2017-02-27 (James Jones)
  - Copy content from VK\_NV\_dedicated\_allocation
  - Add some references to external object interactions to the overview.
- Revision 2, 2017-03-27 (Jason Ekstrand)
  - Rework the extension to be query-based
- Revision 3, 2017-07-31 (Jason Ekstrand)
  - Clarify that memory objects created with VkMemoryDedicatedAllocateInfoKHR can only have the specified resource bound and no others.

## VK\_KHR\_descriptor\_update\_template

### Name String

`VK_KHR_descriptor_update_template`

### Extension Type

Device extension

### Registered Extension Number

86

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- Markus Tavenrath [@mtavenrath](#)

### Last Modified Date

2017-09-05

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Interacts with [VK\\_KHR\\_push\\_descriptor](#)
- Promoted to Vulkan 1.1 Core

### Contributors

- Jeff Bolz, NVIDIA

- Michael Worcester, Imagination Technologies

Applications may wish to update a fixed set of descriptors in a large number of descriptors sets very frequently, i.e. during initializaton phase or if it is required to rebuild descriptor sets for each frame. For those cases it is also not unlikely that all information required to update a single descriptor set is stored in a single struct. This extension provides a way to update a fixed set of descriptors in a single [VkDescriptorSet](#) with a pointer to a user defined data structure describing the new descriptors.

## New Object Types

- [VkDescriptorUpdateTemplateKHR](#)

## New Enum Constants

Extending [VkStructureType](#):

- [VK\\_STRUCTURE\\_TYPE\\_DESCRIPTOR\\_UPDATE\\_TEMPLATE\\_CREATE\\_INFO\\_KHR](#)

## New Enums

- [VkDescriptorUpdateTemplateCreateFlagsKHR](#)
- [VkDescriptorUpdateTemplateTypeKHR](#)

## New Structures

- [VkDescriptorUpdateTemplateEntryKHR](#)
- [VkDescriptorUpdateTemplateCreateInfoKHR](#)

## New Functions

- [vkCreateDescriptorUpdateTemplateKHR](#)
- [vkDestroyDescriptorUpdateTemplateKHR](#)
- [vkUpdateDescriptorSetWithTemplateKHR](#)
- [vkCmdPushDescriptorSetWithTemplateKHR](#)

## Promotion to Vulkan 1.1

[vkCmdPushDescriptorSetWithTemplateKHR](#) is included as an interaction with [VK\\_KHR\\_push\\_descriptor](#). If Vulkan 1.1 and [VK\\_KHR\\_push\\_descriptor](#) are supported, this is included by [VK\\_KHR\\_push\\_descriptor](#).

The base functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Version History

- Revision 1, 2016-01-11 (Markus Tavenrath)

- Initial draft

## **VK\_KHR\_device\_group**

### **Name String**

`VK_KHR_device_group`

### **Extension Type**

Device extension

### **Registered Extension Number**

61

### **Revision**

4

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_device\\_group\\_creation](#)

### **Deprecation state**

- *Promoted* to [Vulkan 1.1](#)

### **Contact**

- Jeff Bolz [@jeffbolz\\_nv](#)

### **Last Modified Date**

2017-10-10

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

- Promoted to Vulkan 1.1 Core

### **Contributors**

- Jeff Bolz, NVIDIA
- Tobias Hector, Imagination Technologies

This extension provides functionality to use a logical device that consists of multiple physical devices, as created with the [VK\\_KHR\\_device\\_group\\_creation](#) extension. A device group can allocate memory across the subdevices, bind memory from one subdevice to a resource on another subdevice, record command buffers where some work executes on an arbitrary subset of the subdevices, and potentially present a swapchain image from one or more subdevices.

## **New Object Types**

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR`
  - `VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR`
  - `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO_KHR`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO_KHR`
- Extending [VkImageCreateFlagBits](#)
  - `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`
- Extending [VkPipelineCreateFlagBits](#)
  - `VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT_KHR`
  - `VK_PIPELINE_CREATE_DISPATCH_BASE_KHR`
- Extending [VkDependencyFlagBits](#)
  - `VK_DEPENDENCY_DEVICE_GROUP_BIT_KHR`
- Extending [VkSwapchainCreateFlagBitsKHR](#)
  - `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`

## New Enums

- [VkPeerMemoryFeatureFlagBitsKHR](#)
- [VkMemoryAllocateFlagBitsKHR](#)
- [VkDeviceGroupPresentModeFlagBitsKHR](#)

## New Structures

- [VkMemoryAllocateFlagsInfoKHR](#)
- [VkDeviceGroupRenderPassBeginInfoKHR](#)
- [VkDeviceGroupCommandBufferBeginInfoKHR](#)
- [VkDeviceGroupSubmitInfoKHR](#)
- [VkDeviceGroupBindSparseInfoKHR](#)
- [VkBindBufferMemoryDeviceGroupInfoKHR](#)
- [VkBindImageMemoryDeviceGroupInfoKHR](#)

- [VkDeviceGroupPresentCapabilitiesKHR](#)
- [VkImageSwapchainCreateInfoKHR](#)
- [VkBindImageMemorySwapchainInfoKHR](#)
- [VkAcquireNextImageInfoKHR](#)
- [VkDeviceGroupPresentInfoKHR](#)
- [VkDeviceGroupSwapchainCreateInfoKHR](#)

## New Functions

- [vkGetDeviceGroupPeerMemoryFeaturesKHR](#)
- [vkCmdSetDeviceMaskKHR](#)
- [vkCmdDispatchBaseKHR](#)
- [vkGetDeviceGroupPresentCapabilitiesKHR](#)
- [vkGetDeviceGroupSurfacePresentModesKHR](#)
- [vkGetPhysicalDevicePresentRectanglesKHR](#)
- [vkAcquireNextImage2KHR](#)

## New Built-In Variables

- `DeviceIndex`

## New SPIR-V Capabilities

- `DeviceGroup`

## Promotion to Vulkan 1.1

The following enums, types and commands are included as interactions with [VK\\_KHR\\_swapchain](#):

- `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR`
- `VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR`
- `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR`
- `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR`
- `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR`
- `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`
- `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`
- [VkDeviceGroupPresentModeFlagBitsKHR](#)
- [VkDeviceGroupPresentCapabilitiesKHR](#)
- [VkImageSwapchainCreateInfoKHR](#)
- [VkBindImageMemorySwapchainInfoKHR](#)
- [VkAcquireNextImageInfoKHR](#)
- [VkDeviceGroupPresentInfoKHR](#)

- [VkDeviceGroupSwapchainCreateInfoKHR](#)
- [vkGetDeviceGroupPresentCapabilitiesKHR](#)
- [vkGetDeviceGroupSurfacePresentModesKHR](#)
- [vkGetPhysicalDevicePresentRectanglesKHR](#)
- [vkAcquireNextImage2KHR](#)

If Vulkan 1.1 and VK\_KHR\_swapchain are supported, these are included by VK\_KHR\_swapchain.

The base functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Examples

TODO

## Version History

- Revision 1, 2016-10-19 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2017-05-19 (Tobias Hector)
  - Removed extended memory bind functions to VK\_KHR\_bind\_memory2, added dependency on that extension, and device-group-specific structs for those functions.
- Revision 3, 2017-10-06 (Ian Elliott)
  - Corrected Vulkan 1.1 interactions with the WSI extensions. All Vulkan 1.1 WSI interactions are with the VK\_KHR\_swapchain extension.
- Revision 4, 2017-10-10 (Jeff Bolz)
  - Rename "SFR" bits and structure members to use the phrase "split instance bind regions".

## VK\_KHR\_device\_group\_creation

### Name String

VK\_KHR\_device\_group\_creation

### Extension Type

Instance extension

### Registered Extension Number

71

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Deprecation state

- *Promoted* to [Vulkan 1.1](#)

## Contact

- Jeff Bolz [@jeffbolznv](#)

## Last Modified Date

2016-10-19

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jeff Bolz, NVIDIA

This extension provides instance-level commands to enumerate groups of physical devices, and to create a logical device from a subset of one of those groups. Such a logical device can then be used with new features in the [VK\\_KHR\\_device\\_group](#) extension.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHR`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO_KHR`
- Extending [VkMemoryHeapFlagBits](#)
  - `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHR`

## New Enums

None.

## New Structures

- [VkPhysicalDeviceGroupPropertiesKHR](#)
- [VkDeviceGroupDeviceCreateInfoKHR](#)

## New Functions

- [vkEnumeratePhysicalDeviceGroupsKHR](#)

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Examples

```
VkDeviceCreateInfo devCreateInfo = { VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO };
// (not shown) fill out devCreateInfo as usual.
uint32_t deviceGroupCount = 0;
VkPhysicalDeviceGroupPropertiesKHR *props = NULL;

// Query the number of device groups
vkEnumeratePhysicalDeviceGroupsKHR(g_vkInstance, &deviceGroupCount, NULL);

// Allocate and initialize structures to query the device groups
props = (VkPhysicalDeviceGroupPropertiesKHR *)malloc(deviceGroupCount*sizeof(VkPhysicalDeviceGroupPropertiesKHR));
for (i = 0; i < deviceGroupCount; ++i) {
    props[i].sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHR;
    props[i].pNext = NULL;
}
vkEnumeratePhysicalDeviceGroupsKHR(g_vkInstance, &deviceGroupCount, props);

// If the first device group has more than one physical device. create
// a logical device using all of the physical devices.
VkDeviceGroupDeviceCreateInfoKHR deviceGroupInfo = {
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO_KHR };
if (props[0].physicalDeviceCount > 1) {
    deviceGroupInfo.physicalDeviceCount = props[0].physicalDeviceCount;
    deviceGroupInfo.pPhysicalDevices = props[0].physicalDevices;
    devCreateInfo.pNext = &deviceGroupInfo;
}

vkCreateDevice(props[0].physicalDevices[0], &devCreateInfo, NULL, &g_vkDevice);
free(props);
```

## Version History

- Revision 1, 2016-10-19 (Jeff Bolz)

- Internal revisions

## **VK\_KHR\_external\_fence**

### **Name String**

`VK_KHR_external_fence`

### **Extension Type**

Device extension

### **Registered Extension Number**

114

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_fence\\_capabilities](#)

### **Deprecation state**

- *Promoted* to [Vulkan 1.1](#)

### **Contact**

- Jesse Hall [critsec](#)

### **Last Modified Date**

2017-05-08

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

- Promoted to Vulkan 1.1 Core

### **Contributors**

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Cass Everitt, Oculus
- Contributors to [VK\\_KHR\\_external\\_semaphore](#)

An application using external memory may wish to synchronize access to that memory using fences. This extension enables an application to create fences from which non-Vulkan handles that reference the underlying synchronization primitive can be exported.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO_KHR`

## New Enums

- `VkFenceImportFlagBitsKHR`

## New Structs

- `VkExportFenceCreateInfoKHR`

## New Functions

None.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

This extension borrows concepts, semantics, and language from `VK_KHR_external_semaphore`. That extension's issues apply equally to this extension.

## Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial revision

## `VK_KHR_external_fence_capabilities`

### Name String

`VK_KHR_external_fence_capabilities`

### Extension Type

Instance extension

### Registered Extension Number

113

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Deprecation state

- *Promoted* to [Vulkan 1.1](#)

## Contact

- Jesse Hall [@critsec](#)

## Last Modified Date

2017-05-08

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Cass Everitt, Oculus
- Contributors to [VK\\_KHR\\_external\\_semaphore\\_capabilities](#)

An application may wish to reference device fences in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension provides a set of capability queries and handle definitions that allow an application to determine what types of “external” fence handles an implementation supports for a given set of use cases.

## New Object Types

None.

## New Enum Constants

- [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_EXTERNAL\\_FENCE\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_EXTERNAL\\_FENCE\\_PROPERTIES\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_ID\\_PROPERTIES\\_KHR](#)
- [VK\\_LUID\\_SIZE\\_KHR](#)

## New Enums

- [VkExternalFenceHandleTypeFlagBitsKHR](#)
- [VkExternalFenceFeatureFlagBitsKHR](#)

## New Structs

- [VkPhysicalDeviceExternalFenceInfoKHR](#)
- [VkExternalFencePropertiesKHR](#)
- [VkPhysicalDeviceIDPropertiesKHR](#)

## New Functions

- [vkGetPhysicalDeviceExternalFencePropertiesKHR](#)

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial version

## VK\_KHR\_external\_memory

### Name String

`VK_KHR_external_memory`

### Extension Type

Device extension

### Registered Extension Number

73

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_external\\_memory\\_capabilities](#)

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- James Jones [Qcubanismo](#)

## Last Modified Date

2016-10-20

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Interacts with [VK\\_KHR\\_dedicated\\_allocation](#).
- Interacts with [VK\\_NV\\_dedicated\\_allocation](#).
- Promoted to Vulkan 1.1 Core

## Contributors

- Jason Ekstrand, Intel
- Ian Elliot, Google
- Jesse Hall, Google
- Tobias Hector, Imagination Technologies
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Matthew Netsch, Qualcomm Technologies, Inc.
- Daniel Rakos, AMD
- Carsten Rohde, NVIDIA
- Ray Smith, ARM
- Chad Versace, Google

An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension enables an application to export non-Vulkan handles from Vulkan memory objects such that the underlying resources can be referenced outside the scope of the Vulkan logical device that created them.

## New Object Types

None.

## New Enum Constants

- [VK\\_STRUCTURE\\_TYPE\\_EXTERNAL\\_MEMORY\\_BUFFER\\_CREATE\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_EXTERNAL\\_MEMORY\\_IMAGE\\_CREATE\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_EXPORT\\_MEMORY\\_ALLOCATE\\_INFO\\_KHR](#)
- [VK\\_QUEUE\\_FAMILY\\_EXTERNAL\\_KHR](#)
- [VK\\_ERROR\\_INVALID\\_EXTERNAL\\_HANDLE\\_KHR](#)

## New Enums

None.

## New Structs

- [VkExternalMemoryImageCreateInfoKHR](#)
- [VkExternalMemoryBufferCreateInfoKHR](#)
- [VkExportMemoryAllocateInfoKHR](#)

## New Functions

None.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

1) How do applications correlate two physical devices across process or Vulkan instance boundaries?

**RESOLVED:** New device ID fields have been introduced by [VK\\_KHR\\_external\\_memory\\_capabilities](#). These fields, combined with the existing [VkPhysicalDeviceProperties::driverVersion](#) field can be used to identify compatible devices across processes, drivers, and APIs. [VkPhysicalDeviceProperties::pipelineCacheUUID](#) is not sufficient for this purpose because despite its description in the specification, it need only identify a unique pipeline cache format in practice. Multiple devices may be able to use the same pipeline cache data, and hence it would be desirable for all of them to have the same pipeline cache UUID. However, only the same concrete physical device can be used when sharing memory, so an actual unique device ID was introduced. Further, the pipeline cache UUID was specific to Vulkan, but correlation with other, non-extensible APIs is required to enable interoperation with those APIs.

2) If memory objects are shared between processes and APIs, is this considered aliasing according to the rules outlined in the [Memory Aliasing](#) section?

**RESOLVED:** Yes. Applications must take care to obey all restrictions imposed on aliased resources when using memory across multiple Vulkan instances or other APIs.

3) Are new image layouts or metadata required to specify image layouts and layout transitions compatible with non-Vulkan APIs, or with other instances of the same Vulkan driver?

**RESOLVED:** Separate instances of the same Vulkan driver running on the same GPU should have identical internal layout semantics, so applications have the tools they need to ensure views of images are consistent between the two instances. Other APIs will fall into two categories: Those that are Vulkan-compatible, and those that are Vulkan-incompatible. Vulkan-incompatible APIs will require the image to be in the GENERAL layout whenever they are accessing them.

Note this does not attempt to address cross-device transitions, nor transitions to engines on the same device which are not visible within the Vulkan API. Both of these are beyond the scope of this extension.

4) Is a new barrier flag or operation of some type needed to prepare external memory for handoff to another Vulkan instance or API and/or receive it from another instance or API?

**RESOLVED:** Yes. Some implementations need to perform additional cache management when transitioning memory between address spaces, and other APIs, instances, or processes may operate in a separate address space. Options for defining this transition include:

- A new structure that can be added to the `pNext` list in `VkMemoryBarrier`, `VkBufferMemoryBarrier`, and `VkImageMemoryBarrier`.
- A new bit in `VkAccessFlags` that can be set to indicate an “external” access.
- A new bit in `VkDependencyFlags`
- A new special queue family that represents an “external” queue.

A new structure has the advantage that the type of external transition can be described in as much detail as necessary. However, there is not currently a known need for anything beyond differentiating external vs. internal accesses, so this is likely an over-engineered solution. The access flag bit has the advantage that it can be applied at buffer, image, or global granularity, and semantically it maps pretty well to the operation being described. Additionally, the API already includes `VK_ACCESS_MEMORY_READ_BIT` and `VK_ACCESS_MEMORY_WRITE_BIT` which appear to be intended for this purpose. However, there is no obvious pipeline stage that would correspond to an external access, and therefore no clear way to use `VK_ACCESS_MEMORY_READ_BIT` or `VK_ACCESS_MEMORY_WRITE_BIT`. `VkDependencyFlags` and `VkPipelineStageFlags` operate at command granularity rather than image or buffer granularity, which would make an entire pipeline barrier an internal→external or external→internal barrier. This may not be a problem in practice, but seems like the wrong scope. Another downside of `VkDependencyFlags` is that it lacks inherent directionality: There are not `src` and `dst` variants of it in the barrier or dependency description semantics, so two bits might need to be added to describe both internal→external and external→internal transitions. Transitioning a resource to a special queue family corresponds well with the operation of transitioning to a separate Vulkan instance, in that both operations ideally include scheduling a barrier on both sides of the transition: Both the releasing and the acquiring queue or process. Using a special queue family requires adding an additional reserved queue family index. Re-using `VK_QUEUE_FAMILY_IGNORED` would have left it unclear how to transition a concurrent usage resource from one process to another, since the semantics would have likely been equivalent to the currently-ignored transition of `VK_QUEUE_FAMILY_IGNORED` → `VK_QUEUE_FAMILY_IGNORED`. Fortunately, creating a new reserved queue family index is not invasive.

Based on the above analysis, the approach of transitioning to a special “external” queue family was chosen.

5) Do internal driver memory arrangements and/or other internal driver image properties need to be exported and imported when sharing images across processes or APIs.

**RESOLVED:** Some vendors claim this is necessary on their implementations, but it was determined that the security risks of allowing opaque meta data to be passed from applications to the driver

were too high. Therefore, implementations which require metadata will need to associate it with the objects represented by the external handles, and rely on the dedicated allocation mechanism to associate the exported and imported memory objects with a single image or buffer.

6) Most prior interoperation and cross-process sharing APIs have been based on image-level sharing. Should Vulkan sharing be based on memory-object sharing or image sharing?

**RESOLVED:** These extensions have assumed memory-level sharing is the correct granularity. Vulkan is a lower-level API than most prior APIs, and as such attempts to closely align with the underlying primitives of the hardware and system-level drivers it abstracts. In general, the resource that holds the backing store for both images and buffers of various types is memory. Images and buffers are merely metadata containing brief descriptions of the layout of bits within that memory.

Because memory object-based sharing is aligned with the overall Vulkan API design, it exposes the full power of Vulkan on external objects. External memory can be used as backing for sparse images, for example, whereas such usage would be awkward at best with a sharing mechanism based on higher-level primitives such as images. Further, aligning the mechanism with the API in this way provides some hope of trivial compatibility with future API enhancements. If new objects backed by memory objects are added to the API, they too can be used across processes with minimal additions to the base external memory APIs.

Earlier APIs implemented interop at a higher level, and this necessitated entirely separate sharing APIs for images and buffers. To co-exist and interoperate with those APIs, the Vulkan external sharing mechanism must accommodate their model. However, if it can be agreed that memory-based sharing is the more desirable and forward-looking design, legacy interoperation considerations can be considered another reason to favor memory-based sharing: While native and legacy driver primitives that may be used to implement sharing may not be as low-level as the API here suggests, raw memory is still the least common denominator among the types. Image-based sharing can be cleanly derived from a set of base memory-object sharing APIs with minimal effort, whereas image-based sharing does not generalize well to buffer or raw-memory sharing. Therefore, following the general Vulkan design principle of minimalism, it is better to expose even interoperability with image-based native and external primitives via the memory sharing API, and place sufficient limits on their usage to ensure they can be used only as backing for equivalent Vulkan images. This provides a consistent API for applications regardless of which platform or external API they are targeting, which makes development of multi-API and multi-platform applications simpler.

7) Should Vulkan define a common external handle type and provide Vulkan functions to facilitate cross-process sharing of such handles rather than relying on native handles to define the external objects?

**RESOLVED:** No. Cross-process sharing of resources is best left to native platforms. There are myriad security and extensibility issues with such a mechanism, and attempting to re-solve all those issues within Vulkan does not align with Vulkan's purpose as a graphics API. If desired, such a mechanism could be built as a layer or helper library on top of the opaque native handle defined in this family of extensions.

8) Must implementations provide additional guarantees about state implicitly included in memory

objects for those memory objects that may be exported?

**RESOLVED:** Implementations must ensure that sharing memory objects does not transfer any information between the exporting and importing instances and APIs other than that required to share the data contained in the memory objects explicitly shared. As specific examples, data from previously freed memory objects that used the same underlying physical memory, and data from memory objects using adjacent physical memory must not be visible to applications importing an exported memory object.

9) Must implementations validate external handles the application provides as input to memory import operations?

**RESOLVED:** Implementations must return an error to the application if the provided memory handle cannot be used to complete the requested import operation. However, implementations need not validate handles are of the exact type specified by the application.

## Version History

- Revision 1, 2016-10-20 (James Jones)
  - Initial version

## VK\_KHR\_external\_memory\_capabilities

### Name String

`VK_KHR_external_memory_capabilities`

### Extension Type

Instance extension

### Registered Extension Number

72

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- James Jones [Qcubanismo](#)

### Last Modified Date

2016-10-17

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Interacts with [VK\\_KHR\\_dedicated\\_allocation](#).
- Interacts with [VK\\_NV\\_dedicated\\_allocation](#).
- Promoted to Vulkan 1.1 Core

## Contributors

- Ian Elliot, Google
- Jesse Hall, Google
- James Jones, NVIDIA

An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension provides a set of capability queries and handle definitions that allow an application to determine what types of “external” memory handles an implementation supports for a given set of use cases.

## New Object Types

None.

## New Enum Constants

- [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_EXTERNAL\\_IMAGE\\_FORMAT\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_EXTERNAL\\_IMAGE\\_FORMAT\\_PROPERTIES\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_EXTERNAL\\_BUFFER\\_INFO\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_EXTERNAL\\_BUFFER\\_PROPERTIES\\_KHR](#)
- [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_ID\\_PROPERTIES\\_KHR](#)
- [VK\\_LUID\\_SIZE\\_KHR](#)

## New Enums

- [VkExternalMemoryHandleTypeFlagBitsKHR](#)
- [VkExternalMemoryFeatureFlagBitsKHR](#)

## New Structs

- [VkExternalMemoryPropertiesKHR](#)
- [VkPhysicalDeviceExternalImageFormatInfoKHR](#)
- [VkExternalImageFormatPropertiesKHR](#)
- [VkPhysicalDeviceExternalBufferInfoKHR](#)
- [VkExternalBufferPropertiesKHR](#)
- [VkPhysicalDeviceIDPropertiesKHR](#)

## New Functions

- [vkGetPhysicalDeviceExternalBufferPropertiesKHR](#)

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

1) Why do so many external memory capabilities need to be queried on a per-memory-handle-type basis?

**PROPOSED RESOLUTION:** This is because some handle types are based on OS-native objects that have far more limited capabilities than the very generic Vulkan memory objects. Not all memory handle types can name memory objects that support 3D images, for example. Some handle types cannot even support the deferred image and memory binding behavior of Vulkan and require specifying the image when allocating or importing the memory object.

2) Do the [VkExternalImageFormatPropertiesKHR](#) and [VkExternalBufferPropertiesKHR](#) structs need to include a list of memory type bits that support the given handle type?

**PROPOSED RESOLUTION:** No. The memory types that don't support the handle types will simply be filtered out of the results returned by [vkGetImageMemoryRequirements](#) and [vkGetBufferMemoryRequirements](#) when a set of handle types was specified at image or buffer creation time.

3) Should the non-opaque handle types be moved to their own extension?

**PROPOSED RESOLUTION:** Perhaps. However, defining the handle type bits does very little and does not require any platform-specific types on its own, and it's easier to maintain the bitfield values in a single extension for now. Presumably more handle types could be added by separate extensions though, and it would be mildly weird to have some platform-specific ones defined in the core spec and some in extensions

4) Do we need a [D3D11\\_TILEPOOL](#) type?

**PROPOSED RESOLUTION:** No. This is technically possible, but the synchronization is awkward. D3D11 surfaces must be synchronized using shared mutexes, and these synchronization primitives are shared by the entire memory object, so D3D11 shared allocations divided among multiple buffer and image bindings may be difficult to synchronize.

5) Should the Windows 7-compatible handle types be named “KMT” handles or “GLOBAL\_SHARE” handles?

**PROPOSED RESOLUTION:** KMT, simply because it is more concise.

6) How do applications identify compatible devices and drivers across instance, process, and API boundaries when sharing memory?

**PROPOSED RESOLUTION:** New device properties are exposed that allow applications to correctly correlate devices and drivers. A device and driver UUID that must both match to ensure sharing compatibility between two Vulkan instances, or a Vulkan instance and an extensible external API are added. To allow correlating with Direct3D devices, a device LUID is added that corresponds to a DXGI adapter LUID. A driver ID is not needed for Direct3D because mismatched driver component versions are not a currently supported configuration on the Windows OS. Should support for such configurations be introduced at the OS level, further Vulkan extensions would be needed to correlate userspace component builds.

## Version History

- Revision 1, 2016-10-17 (James Jones)
  - Initial version

## VK\_KHR\_external\_semaphore

### Name String

`VK_KHR_external_semaphore`

### Extension Type

Device extension

### Registered Extension Number

78

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_external_semaphore_capabilities`

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- James Jones [@cubanismo](#)

### Last Modified Date

2016-10-21

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jason Ekstrand, Intel
- Jesse Hall, Google
- Tobias Hector, Imagination Technologies
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Matthew Netsch, Qualcomm Technologies, Inc.
- Ray Smith, ARM
- Chad Versace, Google

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to create semaphores from which non-Vulkan handles that reference the underlying synchronization primitive can be exported.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHR`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR`

## New Enums

- `VkSemaphoreImportFlagBitsKHR`

## New Structs

- `VkExportSemaphoreCreateInfoKHR`

## New Functions

None.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

1) Should there be restrictions on what side effects can occur when waiting on imported semaphores that are in an invalid state?

**RESOLVED:** Yes. Normally, validating such state would be the responsibility of the application, and

the implementation would be free to enter an undefined state if valid usage rules were violated. However, this could cause security concerns when using imported semaphores, as it would require the importing application to trust the exporting application to ensure the state is valid. Requiring this level of trust is undesirable for many potential use cases.

2) Must implementations validate external handles the application provides as input to semaphore state import operations?

**RESOLVED:** Implementations must return an error to the application if the provided semaphore state handle cannot be used to complete the requested import operation. However, implementations need not validate handles are of the exact type specified by the application.

## Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

## **VK\_KHR\_external\_semaphore\_capabilities**

### Name String

`VK_KHR_external_semaphore_capabilities`

### Extension Type

Instance extension

### Registered Extension Number

77

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- James Jones [Qcubanismo](#)

### Last Modified Date

2016-10-20

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA

An application may wish to reference device semaphores in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension provides a set of capability queries and handle definitions that allow an application to determine what types of “external” semaphore handles an implementation supports for a given set of use cases.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO_KHR`
- `VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES_KHR`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHR`
- `VK_LUID_SIZE_KHR`

## New Enums

- `VkExternalSemaphoreHandleTypeFlagBitsKHR`
- `VkExternalSemaphoreFeatureFlagBitsKHR`

## New Structs

- `VkPhysicalDeviceExternalSemaphoreInfoKHR`
- `VkExternalSemaphorePropertiesKHR`
- `VkPhysicalDeviceIDPropertiesKHR`

## New Functions

- `vkGetPhysicalDeviceExternalSemaphorePropertiesKHR`

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

## Version History

- Revision 1, 2016-10-20 (James Jones)
  - Initial revision

## VK\_KHR\_get\_memory\_requirements2

### Name String

`VK_KHR_get_memory_requirements2`

### Extension Type

Device extension

### Registered Extension Number

147

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- Jason Ekstrand [@jekstrand](#)

### Last Modified Date

2017-09-05

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

### Contributors

- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA
- Jesse Hall, Google

This extension provides new entry points to query memory requirements of images and buffers in a way that can be easily extended by other extensions, without introducing any further entry points. The Vulkan 1.0 `VkMemoryRequirements` and `VkSparseImageMemoryRequirements` structures do not include a `sType/pNext`, this extension wraps them in new structures with `sType`/`pNext` so an application can query a chain of memory requirements structures by constructing the chain and letting the implementation fill them in. A new command is added for each

`vkGet*MemoryRequirements` command in core Vulkan 1.0.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2_KHR`
  - `VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2_KHR`
  - `VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2_KHR`
  - `VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2_KHR`
  - `VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2_KHR`

## New Enums

None.

## New Structures

- `VkBufferMemoryRequirementsInfo2KHR`
- `VkImageMemoryRequirementsInfo2KHR`
- `VkImageSparseMemoryRequirementsInfo2KHR`
- `VkMemoryRequirements2KHR`
- `VkSparseImageMemoryRequirements2KHR`

## New Functions

- `vkGetImageMemoryRequirements2KHR`
- `vkGetBufferMemoryRequirements2KHR`
- `vkGetImageSparseMemoryRequirements2KHR`

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Version History

- Revision 1, 2017-03-23 (Jason Ekstrand)

- Internal revisions

## **VK\_KHR\_get\_physical\_device\_properties2**

### **Name String**

`VK_KHR_get_physical_device_properties2`

### **Extension Type**

Instance extension

### **Registered Extension Number**

60

### **Revision**

2

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Deprecation state**

- *Promoted* to [Vulkan 1.1](#)

### **Contact**

- Jeff Bolz [@jeffbolznv](#)

### **Last Modified Date**

2017-09-05

### **IP Status**

No known IP claims.

### **Interactions and External Dependencies**

- Promoted to Vulkan 1.1 Core

### **Contributors**

- Jeff Bolz, NVIDIA
- Ian Elliott, Google

This extension provides new entry points to query device features, device properties, and format properties in a way that can be easily extended by other extensions, without introducing any further entry points. The Vulkan 1.0 feature/limit/formatproperty structures do not include `sType`/`pNext` members. This extension wraps them in new structures with `sType`/`pNext` members, so an application can query a chain of feature/limit/formatproperty structures by constructing the chain and letting the implementation fill them in. A new command is added for each `vkGetPhysicalDevice*` command in core Vulkan 1.0. The new feature structure (and a chain of extension structures) can also be passed in to device creation to enable features.

This extension also allows applications to use the physical-device components of device extensions

before `vkCreateDevice` is called.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2_KHR`
  - `VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2_KHR`
  - `VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2_KHR`
  - `VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2_KHR`
  - `VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2_KHR`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceFeatures2KHR`
- `VkPhysicalDeviceProperties2KHR`
- `VkFormatProperties2KHR`
- `VkImageFormatProperties2KHR`
- `VkPhysicalDeviceImageFormatInfo2KHR`
- `VkQueueFamilyProperties2KHR`
- `VkPhysicalDeviceMemoryProperties2KHR`
- `VkSparseImageFormatProperties2KHR`
- `VkPhysicalDeviceSparseImageFormatInfo2KHR`

## New Functions

- `vkGetPhysicalDeviceFeatures2KHR`
- `vkGetPhysicalDeviceProperties2KHR`
- `vkGetPhysicalDeviceFormatProperties2KHR`
- `vkGetPhysicalDeviceImageFormatProperties2KHR`
- `vkGetPhysicalDeviceQueueFamilyProperties2KHR`

- [vkGetPhysicalDeviceMemoryProperties2KHR](#)
- [vkGetPhysicalDeviceSparseImageFormatProperties2KHR](#)

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Examples

```
// Get features with a hypothetical future extension.
VkHypotheticalExtensionFeaturesKHR hypotheticalFeatures =
{
    VK_STRUCTURE_TYPE_HYPOTHETICAL_FEATURES_KHR, // sType
    NULL, // pNext
};

VkPhysicalDeviceFeatures2KHR features =
{
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR, // sType
    &hypotheticalFeatures, // pNext
};

// After this call, features and hypotheticalFeatures have been filled out.
vkGetPhysicalDeviceFeatures2KHR(physicalDevice, &features);

// Properties/limits can be chained and queried similarly.

// Enable some features:
VkHypotheticalExtensionFeaturesKHR enabledHypotheticalFeatures =
{
    VK_STRUCTURE_TYPE_HYPOTHETICAL_FEATURES_KHR, // sType
    NULL, // pNext
};

VkPhysicalDeviceFeatures2KHR enabledFeatures =
{
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR, // sType
};
```

```

    &enabledHypotheticalFeatures,                                //  

pNext  

};  
  

    enabledFeatures.features.xyz = VK_TRUE;  

    enabledHypotheticalFeatures.abc = VK_TRUE;  
  

    VkDeviceCreateInfo deviceCreateInfo =  

    {  

        VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,                      //  

sType  

        &enabledFeatures,                                         //  

pNext  

        ...  

        NULL,                                                 //  

pEnabledFeatures  

    }  
  

    VkDevice device;  

    vkCreateDevice(physicalDevice, &deviceCreateInfo, NULL, &device);

```

## Version History

- Revision 1, 2016-09-12 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2016-11-02 (Ian Elliott)
  - Added ability for applications to use the physical-device components of device extensions before `vkCreateDevice` is called.

## VK\_KHR\_maintenance1

### Name String

`VK_KHR_maintenance1`

### Extension Type

Device extension

### Registered Extension Number

70

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

## Contact

- Piers Daniell [Opdaniell-nv](#)

## Last Modified Date

2018-03-13

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Dan Ginsburg, Valve
- Daniel Koch, NVIDIA
- Daniel Rakos, AMD
- Jan-Harald Fredriksen, ARM
- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA
- Jesse Hall, Google
- John Kessenich, Google
- Michael Worcester, Imagination Technologies
- Neil Henning, Codeplay Software Ltd.
- Piers Daniell, NVIDIA
- Slawomir Grajewski, Intel
- Tobias Hector, Imagination Technologies
- Tom Olson, ARM

`VK_KHR_maintenance1` adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release.

The new features are as follows:

- Allow 2D and 2D array image views to be created from 3D images, which can then be used as color framebuffer attachments. This allows applications to render to slices of a 3D image.
- Support `vkCmdCopyImage` between 2D array layers and 3D slices. This extension allows copying from layers of a 2D array image to slices of a 3D image and vice versa.
- Allow negative height to be specified in the `VkViewport::height` field to perform y-inversion of the clip-space to framebuffer-space transform. This allows apps to avoid having to use `gl_Position.y = -gl_Position.y` in shaders also targeting other APIs.
- Allow implementations to express support for doing just transfers and clears of image formats that they otherwise support no other format features for. This is done by adding new format feature flags `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR` and `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`.
- Support `vkCmdFillBuffer` on transfer-only queues. Previously `vkCmdFillBuffer` was defined to

only work on command buffers allocated from command pools which support graphics or compute queues. It is now allowed on queues that just support transfer operations.

- Fix the inconsistency of how error conditions are returned between the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` functions and the `vkAllocateDescriptorSets` and `vkAllocateCommandBuffers` functions.
- Add new `VK_ERROR_OUT_OF_POOL_MEMORY_KHR` error so implementations can give a more precise reason for `vkAllocateDescriptorSets` failures.
- Add a new command `vkTrimCommandPoolKHR` which gives the implementation an opportunity to release any unused command pool memory back to the system.

## New Object Types

None.

## New Enum Constants

- `VK_ERROR_OUT_OF_POOL_MEMORY_KHR`
- `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR`
- `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`
- `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT_KHR`

## New Enums

None.

## New Structures

None.

## New Functions

- `vkTrimCommandPoolKHR`

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

1. Are viewports with zero height allowed?

**RESOLVED:** Yes, although they have low utility.

## Version History

- Revision 1, 2016-10-26 (Piers Daniell)

- Internal revisions
- Revision 2, 2018-03-13 (Jon Leech)
  - Add issue for zero-height viewports

## **VK\_KHR\_maintenance2**

### **Name String**

[VK\\_KHR\\_maintenance2](#)

### **Extension Type**

Device extension

### **Registered Extension Number**

118

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Deprecation state**

- *Promoted* to [Vulkan 1.1](#)

### **Contact**

- Michael Worcester [@michaelworcester](#)

### **Last Modified Date**

2017-09-05

### **Interactions and External Dependencies**

- Promoted to Vulkan 1.1 Core

### **Contributors**

- Michael Worcester, Imagination Technologies
- Stuart Smith, Imagination Technologies
- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA
- Jan-Harald Fredriksen, ARM
- Daniel Rakos, AMD
- Neil Henning, Codeplay
- Piers Daniell, NVIDIA

[VK\\_KHR\\_maintenance2](#) adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release.

The new features are as follows:

- Allow the application to specify which aspect of an input attachment might be read for a given subpass.
- Allow implementations to express the clipping behavior of points.
- Allow creating images with usage flags that may not be supported for the base image's format, but are supported for image views of the image that have a different but compatible format.
- Allow creating uncompressed image views of compressed images.
- Allow the application to select between an upper-left and lower-left origin for the tessellation domain space.
- Adds two new image layouts for depth stencil images to allow either the depth or stencil aspect to be read-only while the other aspect is writable.

## Input Attachment Specification

Input attachment specification allows an application to specify which aspect of a multi-aspect image (e.g. a combined depth stencil format) will be accessed via a `subpassLoad` operation.

On some implementations there **may** be a performance penalty if the implementation does not know (at `vkCreateRenderPass` time) which aspect(s) of multi-aspect images **can** be accessed as input attachments.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES_KHR`
  - `VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO_KHR`
- Extending `VkImageCreateFlagBits`:
  - `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT_KHR`
  - `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR`
- Extending `VkImageLayout`
  - `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL_KHR`
  - `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL_KHR`
- `VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES_KHR`
- `VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY_KHR`

## New Enums

- [VkPointClippingBehaviorKHR](#)
- [VkTessellationDomainOriginKHR](#)

## New Structures

- [VkPhysicalDevicePointClippingPropertiesKHR](#)
- [VkRenderPassInputAttachmentAspectCreateInfoKHR](#)
- [VkInputAttachmentAspectReferenceKHR](#)
- [VkImageViewUsageCreateInfoKHR](#)
- [VkPipelineTessellationDomainOriginStateCreateInfoKHR](#)

## New Functions

None.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Input Attachment Specification Example

Consider the case where a render pass has two subpasses and two attachments.

Attachment 0 has the format `VK_FORMAT_D24_UNORM_S8_UINT`, attachment 1 has some color format.

Subpass 0 writes to attachment 0, subpass 1 reads only the depth information from attachment 0 (using `inputAttachmentRead`) and writes to attachment 1.

```

VkInputAttachmentAspectReferenceKHR references[] = {
    {
        .subpass = 1,
        .inputAttachmentIndex = 0,
        .aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT
    }
};

VkRenderPassInputAttachmentAspectCreateInfoKHR specifyAspects = {
    .sType =
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO_KHR,
    .pNext = NULL,
    .aspectReferenceCount = 1,
    .pAspectReferences = references
};

VkRenderPassCreateInfo createInfo = {
    ...
    .pNext = &specifyAspects,
    ...
}

vkCreateRenderPass(...);

```

## Issues

- 1) What is the default tessellation domain origin?

**RESOLVED:** Vulkan 1.0 originally inadvertently documented a lower-left origin, but the conformance tests and all implementations implemented an upper-left origin. This extension adds a control to select between lower-left (for compatibility with OpenGL) and upper-left, and we retroactively fix unextended Vulkan to have a default of an upper-left origin.

## Version History

- Revision 1, 2017-04-28

## VK\_KHR\_maintenance3

### Name String

VK\_KHR\_maintenance3

### Extension Type

Device extension

### Registered Extension Number

169

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Deprecation state

- *Promoted* to [Vulkan 1.1](#)

## Contact

- Jeff Bolz [@jeffbolznv](#)

## Status

Draft

## Last Modified Date

2017-09-05

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jeff Bolz, NVIDIA

[VK\\_KHR\\_maintenance3](#) adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release.

The new features are as follows:

- A limit on the maximum number of descriptors that are supported in a single descriptor set layout. Some implementations have a limit on the total size of descriptors in a set, which cannot be expressed in terms of the limits in Vulkan 1.0.
- A limit on the maximum size of a single memory allocation. Some platforms have kernel interfaces that limit the maximum size of an allocation.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_MAINTENANCE\\_3\\_PROPERTIES\\_KHR](#)
  - [VK\\_STRUCTURE\\_TYPE\\_DESCRIPTOR\\_SET\\_LAYOUT\\_SUPPORT\\_KHR](#)

## New Enums

None.

## New Structures

- [VkPhysicalDeviceMaintenance3PropertiesKHR](#)
- [VkDescriptorSetLayoutSupportKHR](#)

## New Functions

- [vkGetDescriptorSetLayoutSupportKHR](#)

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Version History

- Revision 1, 2017-08-22

## VK\_KHR\_multiview

### Name String

`VK_KHR_multiview`

### Extension Type

Device extension

### Registered Extension Number

54

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- Jeff Bolz [@jeffbolznv](#)

## Last Modified Date

2016-10-28

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Jeff Bolz, NVIDIA

This extension has the same goal as the OpenGL ES `GL_OVR_multiview` extension - it enables rendering to multiple “views” by recording a single set of commands to be executed with slightly different behavior for each view. It includes a concise way to declare a render pass with multiple views, and gives implementations freedom to render the views in the most efficient way possible.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES_KHR`
- Extending `VkDependencyFlagBits`
  - `VK_DEPENDENCY_VIEW_LOCAL_BIT_KHR`

## New Enums

None.

## New Structures

- `VkPhysicalDeviceMultiviewFeaturesKHR`
- `VkPhysicalDeviceMultiviewPropertiesKHR`
- `VkRenderPassMultiviewCreateInfoKHR`

## New Functions

None.

## New Built-In Variables

- [ViewIndex](#)

## New SPIR-V Capabilities

- [MultiView](#)

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2016-10-28 (Jeff Bolz)
  - Internal revisions

## VK\_KHR\_relaxed\_block\_layout

### Name String

`VK_KHR_relaxed_block_layout`

### Extension Type

Device extension

### Registered Extension Number

145

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Promoted to Vulkan 1.1*

### Contact

- John Kessenich [@johnkslang](#)

## Last Modified Date

2017-03-26

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- John Kessenich, Google

The `VK_KHR_relaxed_block_layout` extension allows implementations to indicate they can support more variation in block `Offset` decorations. For example, placing a vector of three floats at an offset of  $16^*N + 4$ .

See [Offset and Stride Assignment](#) for details.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Version History

- Revision 1, 2017-03-26 (JohnK)

## `VK_KHR_sampler_ycbcr_conversion`

### Name String

`VK_KHR_sampler_ycbcr_conversion`

### Extension Type

Device extension

### Registered Extension Number

157

### Revision

14

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_maintenance1`
- Requires `VK_KHR_bind_memory2`
- Requires `VK_KHR_get_memory_requirements2`
- Requires `VK_KHR_get_physical_device_properties2`

## Deprecation state

- *Promoted to Vulkan 1.1*

## Contact

- Andrew Garrard [@fluppeteer](#)

## Last Modified Date

2017-08-11

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

## Contributors

- Andrew Garrard, Samsung Electronics
- Tobias Hector, Imagination Technologies
- James Jones, NVIDIA
- Daniel Koch, NVIDIA
- Daniel Rakos, AMD
- Romain Guy, Google
- Jesse Hall, Google
- Tom Cooksey, ARM Ltd
- Jeff Leger, Qualcomm Technologies, Inc
- Jan-Harald Fredriksen, ARM Ltd
- Jan Outters, Samsung Electronics
- Alon Or-bach, Samsung Electronics
- Michael Worcester, Imagination Technologies
- Jeff Bolz, NVIDIA
- Tony Zlatinski, NVIDIA
- Matthew Netsch, Qualcomm Technologies, Inc

This extension provides the ability to perform specified color space conversions during texture sampling operations. It also adds a selection of multi-planar formats, including the ability to bind memory to the planes of an image collectively or separately.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO_KHR`

- VK\_STRUCTURE\_TYPE\_BIND\_IMAGE\_PLANE\_MEMORY\_INFO\_KHR
  - VK\_STRUCTURE\_TYPE\_IMAGE\_PLANE\_MEMORY\_REQUIREMENTS\_INFO\_KHR
  - VK\_STRUCTURE\_TYPE\_PHYSICAL\_DEVICE\_SAMPLER\_YCBCR\_CONVERSION\_FEATURES\_KHR
- Extending [VkFormat](#):
    - VK\_FORMAT\_G8B8G8R8\_422\_UNORM\_KHR
    - VK\_FORMAT\_B8G8R8G8\_422\_UNORM\_KHR
    - VK\_FORMAT\_G8\_B8\_R8\_3PLANE\_420\_UNORM\_KHR
    - VK\_FORMAT\_G8\_B8R8\_2PLANE\_420\_UNORM\_KHR
    - VK\_FORMAT\_G8\_B8\_R8\_3PLANE\_422\_UNORM\_KHR
    - VK\_FORMAT\_G8\_B8R8\_2PLANE\_422\_UNORM\_KHR
    - VK\_FORMAT\_G8\_B8\_R8\_3PLANE\_444\_UNORM\_KHR
    - VK\_FORMAT\_R10X6\_UNORM\_PACK16\_KHR
    - VK\_FORMAT\_R10X6G10X6\_UNORM\_2PACK16\_KHR
    - VK\_FORMAT\_R10X6G10X6B10X6A10X6\_UNORM\_4PACK16\_KHR
    - VK\_FORMAT\_G10X6B10X6G10X6R10X6\_422\_UNORM\_4PACK16\_KHR
    - VK\_FORMAT\_B10X6G10X6R10X6G10X6\_422\_UNORM\_4PACK16\_KHR
    - VK\_FORMAT\_G10X6\_B10X6\_R10X6\_3PLANE\_420\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G10X6\_B10X6R10X6\_2PLANE\_420\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G10X6\_B10X6\_R10X6\_3PLANE\_422\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G10X6\_B10X6R10X6\_2PLANE\_422\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G10X6\_B10X6\_R10X6\_3PLANE\_444\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_R12X4\_UNORM\_PACK16\_KHR
    - VK\_FORMAT\_R12X4G12X4\_UNORM\_2PACK16\_KHR
    - VK\_FORMAT\_R12X4G12X4B12X4A12X4\_UNORM\_4PACK16\_KHR
    - VK\_FORMAT\_G12X4B12X4G12X4R12X4\_422\_UNORM\_4PACK16\_KHR
    - VK\_FORMAT\_B12X4G12X4R12X4G12X4\_422\_UNORM\_4PACK16\_KHR
    - VK\_FORMAT\_G12X4\_B12X4\_R12X4\_3PLANE\_420\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G12X4\_B12X4R12X4\_2PLANE\_420\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G12X4\_B12X4\_R12X4\_3PLANE\_422\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G12X4\_B12X4R12X4\_2PLANE\_422\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G12X4\_B12X4\_R12X4\_3PLANE\_444\_UNORM\_3PACK16\_KHR
    - VK\_FORMAT\_G16B16G16R16\_422\_UNORM\_KHR
    - VK\_FORMAT\_B16G16R16G16\_422\_UNORM\_KHR
    - VK\_FORMAT\_G16\_B16\_R16\_3PLANE\_420\_UNORM\_KHR
    - VK\_FORMAT\_G16\_B16R16\_2PLANE\_420\_UNORM\_KHR
    - VK\_FORMAT\_G16\_B16\_R16\_3PLANE\_422\_UNORM\_KHR
    - VK\_FORMAT\_G16\_B16R16\_2PLANE\_422\_UNORM\_KHR
    - VK\_FORMAT\_G16\_B16\_R16\_3PLANE\_444\_UNORM\_KHR
  - Extending [VkImageAspectFlagBits](#):
    - VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT\_KHR
    - VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT\_KHR

- `VK_IMAGE_ASPECT_PLANE_2_BIT_KHR`
- Extending `VkImageCreateFlagBits`:
  - `VK_IMAGE_CREATE_DISJOINT_BIT_KHR`
- Extending `VkFormatFeatureFlagBits`:
  - `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT_KHR`
  - `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT_KHR`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT_KHR`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT_KHR`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT_KHR`
  - `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT_KHR`
  - `VK_FORMAT_FEATURE_DISJOINT_BIT_KHR`

## New Enums

- `VkSamplerYcbcrModelConversionKHR`
- `VkSamplerYcbcrRangeKHR`
- `VkChromaLocationKHR`

## New Structures

- `VkSamplerYcbcrConversionCreateInfoKHR`
- `VkSamplerYcbcrConversionCreateInfoKHR`
- `VkBindImagePlaneMemoryInfoKHR`
- `VkImagePlaneMemoryRequirementsInfoKHR`
- `VkPhysicalDeviceSamplerYcbcrConversionFeaturesKHR`
- `VkSamplerYcbcrConversionImageFormatPropertiesKHR`

## New Functions

- `vkCreateSamplerYcbcrConversionKHR`
- `vkDestroySamplerYcbcrConversionKHR`

## New Objects

- `VkSamplerYcbcrConversionKHR`

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

## Version History

- Revision 1, 2017-01-24 (Andrew Garrard)
  - Initial draft
- Revision 2, 2017-01-25 (Andrew Garrard)
  - After initial feedback
- Revision 3, 2017-01-27 (Andrew Garrard)
  - Higher bit depth formats, renaming, swizzle
- Revision 4, 2017-02-22 (Andrew Garrard)
  - Added query function, formats as RGB, clarifications
- Revision 5, 2017-04 (Andrew Garrard)
  - Simplified query and removed output conversions
- Revision 6, 2017-04-24 (Andrew Garrard)
  - Tidying, incorporated new image query, restored transfer functions
- Revision 7, 2017-04-25 (Andrew Garrard)
  - Added cosited option/midpoint requirement for formats, "bypassConversion"
- Revision 8, 2017-04-25 (Andrew Garrard)
  - Simplified further
- Revision 9, 2017-04-27 (Andrew Garrard)
  - Disjoint no more
- Revision 10, 2017-04-28 (Andrew Garrard)
  - Restored disjoint
- Revision 11, 2017-04-29 (Andrew Garrard)
  - Now Ycbcr conversion, and KHR
- Revision 12, 2017-06-06 (Andrew Garrard)
  - Added conversion to image view creation
- Revision 13, 2017-07-13 (Andrew Garrard)
  - Allowed cosited-only chroma samples for formats
- Revision 14, 2017-08-11 (Andrew Garrard)
  - Reflected quantization changes in BT.2100-1

## **VK\_KHR\_shader\_draw\_parameters**

### **Name String**

`VK_KHR_shader_draw_parameters`

### **Extension Type**

Device extension

## Registered Extension Number

64

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Deprecation state

- *Promoted* to [Vulkan 1.1](#)

## Contact

- Daniel Koch [Qdgkoch](#)

## Last Modified Date

2017-09-05

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Requires the `SPV_KHR_shader_draw_parameters` SPIR-V extension.
- Requires `GL_ARB_shader_draw_parameters` for GLSL source languages.
- Promoted to Vulkan 1.1 Core

## Contributors

- Daniel Koch, NVIDIA Corporation
- Jeff Bolz, NVIDIA
- Daniel Rakos, AMD
- Jan-Harald Fredriksen, ARM
- John Kessenich, Google
- Stuart Smith, IMG

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_KHR_shader_draw_parameters`

The extension provides access to three additional built-in shader variables in Vulkan:

- `BaseInstance`, which contains the `firstInstance` parameter passed to draw commands,
- `BaseVertex`, which contains the `firstVertex/vertexOffset` parameter passed to draw commands, and
- `DrawIndex`, which contains the index of the draw call currently being processed from an indirect draw call.

When using GLSL source-based shader languages, the following variables from `GL_ARB_shader_draw_parameters` can map to these SPIR-V built-in decorations:

- `in int gl_BaseInstanceARB;` → `BaseInstance`,
- `in int gl_BaseVertexARB;` → `BaseVertex`, and
- `in int gl_DrawIDARB;` → `DrawIndex`.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

None.

## New Structures

None.

## New Functions

None.

## New Built-In Variables

- `BaseInstance`
- `BaseVertex`
- `DrawIndex`

## New SPIR-V Capabilities

- `DrawParameters`

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, however a [feature bit was added](#) to distinguish whether it is actually available or not.

## Issues

1) Is this the same functionality as `GL_ARB_shader_draw_parameters`?

**RESOLVED:** It's actually a superset as it also adds in support for arrayed drawing commands.

In GL for `GL_ARB_shader_draw_parameters`, `gl_BaseVertexARB` holds the integer value passed to the

parameter to the command that resulted in the current shader invocation. In the case where the command has no `baseVertex` parameter, the value of `gl_BaseVertexARB` is zero. This means that `gl_BaseVertexARB = baseVertex` (for `glDrawElements` commands with `baseVertex`) or 0. In particular there are no `glDrawArrays` commands that take a `baseVertex` parameter.

Now in Vulkan, we have `BaseVertex = vertexOffset` (for indexed drawing commands) or `firstVertex` (for arrayed drawing commands), and so Vulkan's version is really a superset of GL functionality.

## Version History

- Revision 1, 2016-10-05 (Daniel Koch)
  - Internal revisions

## VK\_KHR\_storage\_buffer\_storage\_class

### Name String

`VK_KHR_storage_buffer_storage_class`

### Extension Type

Device extension

### Registered Extension Number

132

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Promoted* to [Vulkan 1.1](#)

### Contact

- Alexander Galazin [@alegal-arm](#)

### Last Modified Date

2017-09-05

### IP Status

No known IP claims.

### Interactions and External Dependencies

- This extension requires the `SPV_KHR_storage_buffer_storage_class` SPIR-V extension.
- Promoted to Vulkan 1.1 Core

### Contributors

- Alexander Galazin, ARM

- David Neto, Google

This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_KHR_storage_buffer_storage_class`

This extension provides a new SPIR-V `StorageBuffer` storage class. A `Block`-decorated object in this class is equivalent to a `BufferBlock`-decorated object in the `Uniform` storage class.

## New Enum Constants

None.

## New Structures

None.

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1.

## Issues

None.

## Version History

- Revision 1, 2017-03-23 (Alexander Galazin)
  - Initial draft

## `VK_KHR_variable_pointers`

### Name String

`VK_KHR_variable_pointers`

### Extension Type

Device extension

### Registered Extension Number

121

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
- Requires `VK_KHR_storage_buffer_storage_class`

## Deprecation state

- Promoted to Vulkan 1.1

## Contact

- Jesse Hall critsec

## Last Modified Date

2017-09-05

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Requires the `SPV_KHR_variable_pointers` SPIR-V extension.
- Promoted to Vulkan 1.1 Core

## Contributors

- John Kessenich, Google
- Neil Henning, Codeplay
- David Neto, Google
- Daniel Koch, Nvidia
- Graeme Leese, Broadcom
- Weifeng Zhang, Qualcomm
- Stephen Clarke, Imagination Technologies
- Jason Ekstrand, Intel
- Jesse Hall, Google

The `VK_KHR_variable_pointers` extension allows implementations to indicate their level of support for the `SPV_KHR_variable_pointers` SPIR-V extension. The SPIR-V extension allows shader modules to use invocation-private pointers into uniform and/or storage buffers, where the pointer values can be dynamic and non-uniform.

The `SPV_KHR_variable_pointers` extension introduces two capabilities. The first, `VariablePointersStorageBuffer`, **must** be supported by all implementations of this extension. The second, `VariablePointers`, is optional.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES_KHR`

## New Structures

- `VkPhysicalDeviceVariablePointersFeaturesKHR`

## New SPIR-V Capabilities

- [VariablePointersStorageBuffer](#)
- [VariablePointers](#)

## Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted, however support for the [variablePointersStorageBuffer](#) feature is made optional. The original type, enum and command names are still available as aliases of the core functionality.

## Issues

1) Do we need an optional property for the SPIR-V [VariablePointersStorageBuffer](#) capability or should it be mandatory when this extension is advertised?

**RESOLVED:** Add it as a distinct feature, but make support mandatory. Adding it as a feature makes the extension easier to include in a future core API version. In the extension, the feature is mandatory, so that presence of the extension guarantees some functionality. When included in a core API version, the feature would be optional.

2) Can support for these capabilities vary between shader stages?

**RESOLVED:** No, if the capability is supported in any stage it must be supported in all stages.

3) Should the capabilities be features or limits?

**RESOLVED:** Features, primarily for consistency with other similar extensions.

## Version History

- Revision 1, 2017-03-14 (Jesse Hall and John Kessenich)
  - Internal revisions

## VK\_EXT\_buffer\_device\_address

### Name String

[VK\\_EXT\\_buffer\\_device\\_address](#)

### Extension Type

Device extension

### Registered Extension Number

245

### Revision

2

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_KHR\\_get\\_physical\\_device\\_properties2](#)

## Deprecation state

- *Deprecated* by [VK\\_KHR\\_buffer\\_device\\_address](#) extension

## Contact

- Jeff Bolz [@jeffbolz\\_nv](#)

## Last Modified Date

2019-01-06

## IP Status

No known IP claims.

## Contributors

- Jeff Bolz, NVIDIA
- Neil Henning, AMD
- Tobias Hector, AMD
- Jason Ekstrand, Intel
- Baldur Karlsson, Valve

This extension allows the application to query a 64-bit buffer device address value for a buffer, which can be used to access the buffer memory via the [PhysicalStorageBufferEXT](#) storage class in the [GL\\_EXT\\_buffer\\_reference](#) GLSL extension and [SPV\\_EXT\\_physical\\_storage\\_buffer](#) SPIR-V extension.

It also allows buffer device addresses to be provided by a trace replay tool, so that it matches the address used when the trace was captured.

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_PHYSICAL\\_DEVICE\\_BUFFER\\_DEVICE\\_ADDRESS\\_FEATURES\\_EXT](#)
  - [VK\\_STRUCTURE\\_TYPE\\_BUFFER\\_DEVICE\\_ADDRESS\\_INFO\\_EXT](#)
  - [VK\\_STRUCTURE\\_TYPE\\_BUFFER\\_DEVICE\\_ADDRESS\\_CREATE\\_INFO\\_EXT](#)
- Extending [VkBufferUsageFlagBits](#):
  - [VK\\_BUFFER\\_USAGE\\_SHADER\\_DEVICE\\_ADDRESS\\_BIT\\_EXT](#)
- Extending [VkBufferCreateFlagBits](#):
  - [VK\\_BUFFER\\_CREATE\\_DEVICE\\_ADDRESS\\_CAPTURE\\_REPLAY\\_BIT\\_EXT](#)
- Extending [VkResult](#):

- `VK_ERROR_INVALID_DEVICE_ADDRESS_EXT`

## New Enums

None

## New Structures

- `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT`
- `VkBufferDeviceAddressCreateInfoEXT`
- `VkBufferDeviceAddressCreateInfoEXT`

## New Functions

- `vkGetBufferDeviceAddressEXT`

## New Built-In Variables

None

## New SPIR-V Capabilities

- `PhysicalStorageBufferAddressesEXT`

## Issues

1) Where is `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_ADDRESS_FEATURES_EXT` and `VkPhysicalDeviceBufferAddressFeaturesEXT`?

**RESOLVED:** They were renamed as `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT` and `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` accordingly for consistency. Even though, the old names can still be found in the generated header files for compatibility.

## Version History

- Revision 1, 2018-11-01 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2019-01-06 (Jon Leech)
  - Minor updates to appendix for publication

## `VK_EXT_debug_marker`

### Name String

`VK_EXT_debug_marker`

### Extension Type

Device extension

## Registered Extension Number

23

## Revision

4

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_EXT\\_debug\\_report](#)

## Deprecation state

- *Promoted* to [VK\\_EXT\\_debug\\_utils](#) extension

## Contact

- Baldur Karlsson [@baldurk](#)

## Last Modified Date

2017-01-31

## IP Status

No known IP claims.

## Contributors

- Baldur Karlsson
- Dan Ginsburg, Valve
- Jon Ashburn, LunarG
- Kyle Spagnoli, NVIDIA

The [VK\\_EXT\\_debug\\_marker](#) extension is a device extension. It introduces concepts of object naming and tagging, for better tracking of Vulkan objects, as well as additional commands for recording annotations of named sections of a workload to aid organization and offline analysis in external tools.

## New Object Types

None

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_DEBUG\\_MARKER\\_OBJECT\\_NAME\\_INFO\\_EXT](#)
  - [VK\\_STRUCTURE\\_TYPE\\_DEBUG\\_MARKER\\_OBJECT\\_TAG\\_INFO\\_EXT](#)
  - [VK\\_STRUCTURE\\_TYPE\\_DEBUG\\_MARKER\\_MARKER\\_INFO\\_EXT](#)

## New Enums

None

## New Structures

- [VkDebugMarkerObjectNameInfoEXT](#)
- [VkDebugMarkerObjectTagInfoEXT](#)
- [VkDebugMarkerMarkerInfoEXT](#)

## New Functions

- [vkDebugMarkerSetObjectTagEXT](#)
- [vkDebugMarkerSetObjectNameEXT](#)
- [vkCmdDebugMarkerBeginEXT](#)
- [vkCmdDebugMarkerEndEXT](#)
- [vkCmdDebugMarkerInsertEXT](#)

## Examples

### Example 1

Associate a name with an image, for easier debugging in external tools or with validation layers that can print a friendly name when referring to objects in error messages.

```

extern VkDevice device;
extern VkImage image;

// Must call extension functions through a function pointer:
PFN_vkDebugMarkerSetObjectNameEXT pfnDebugMarkerSetObjectNameEXT =
(PFN_vkDebugMarkerSetObjectNameEXT)vkGetDeviceProcAddr(device,
"vkDebugMarkerSetObjectNameEXT");

// Set a name on the image
const VkDebugMarkerObjectNameInfoEXT imageNameInfo =
{
    VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_NAME_INFO_EXT, // sType
    NULL, // pNext
    VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_EXT, // objectType
    (uint64_t)image, // object
    "Brick Diffuse Texture", // pObjectName
};

pfnDebugMarkerSetObjectNameEXT(device, &imageNameInfo);

// A subsequent error might print:
// Image 'Brick Diffuse Texture' (0xc0dec0dedeadbeef) is used in a
// command buffer with no memory bound to it.

```

## Example 2

Annotating regions of a workload with naming information so that offline analysis tools can display a more usable visualisation of the commands submitted.

```

extern VkDevice device;
extern VkCommandBuffer commandBuffer;

// Must call extension functions through a function pointer:
PFN_vkCmdDebugMarkerBeginEXT pfnCmdDebugMarkerBeginEXT =
(PFN_vkCmdDebugMarkerBeginEXT)vkGetDeviceProcAddr(device, "vkCmdDebugMarkerBeginEXT");
PFN_vkCmdDebugMarkerEndEXT pfnCmdDebugMarkerEndEXT = (PFN_vkCmdDebugMarkerEndEXT)
vkGetDeviceProcAddr(device, "vkCmdDebugMarkerEndEXT");
PFN_vkCmdDebugMarkerInsertEXT pfnCmdDebugMarkerInsertEXT =
(PFN_vkCmdDebugMarkerInsertEXT)vkGetDeviceProcAddr(device, "vkCmdDebugMarkerInsertEXT");

// Describe the area being rendered
const VkDebugMarkerMarkerInfoEXT houseMarker =
{
    VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT, // sType
    NULL, // pNext
    "Brick House", // pMarkerName
    { 1.0f, 0.0f, 0.0f, 1.0f }, // color
};

```

```

// Start an annotated group of calls under the 'Brick House' name
pfnCmdDebugMarkerBeginEXT(commandBuffer, &houseMarker);
{
    // A mutable structure for each part being rendered
    VkDebugMarkerMarkerInfoEXT housePartMarker =
    {
        VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT, // sType
        NULL, // pNext
        NULL, // pMarkerName
        { 0.0f, 0.0f, 0.0f, 0.0f }, // color
    };

    // Set the name and insert the marker
    housePartMarker.pMarkerName = "Walls";
    pfnCmdDebugMarkerInsertEXT(commandBuffer, &housePartMarker);

    // Insert the drawcall for the walls
    vkCmdDrawIndexed(commandBuffer, 1000, 1, 0, 0, 0);

    // Insert a recursive region for two sets of windows
    housePartMarker.pMarkerName = "Windows";
    pfnCmdDebugMarkerBeginEXT(commandBuffer, &housePartMarker);
    {
        vkCmdDrawIndexed(commandBuffer, 75, 6, 1000, 0, 0);
        vkCmdDrawIndexed(commandBuffer, 100, 2, 1450, 0, 0);
    }
    pfnCmdDebugMarkerEndEXT(commandBuffer);

    housePartMarker.pMarkerName = "Front Door";
    pfnCmdDebugMarkerInsertEXT(commandBuffer, &housePartMarker);

    vkCmdDrawIndexed(commandBuffer, 350, 1, 1650, 0, 0);

    housePartMarker.pMarkerName = "Roof";
    pfnCmdDebugMarkerInsertEXT(commandBuffer, &housePartMarker);

    vkCmdDrawIndexed(commandBuffer, 500, 1, 2000, 0, 0);
}
// End the house annotation started above
pfnCmdDebugMarkerEndEXT(commandBuffer);

```

## Issues

- 1) Should the tag or name for an object be specified using the `pNext` parameter in the object's `VkCreateInfo` structure?

**RESOLVED:** No. While this fits with other Vulkan patterns and would allow more type safety and future proofing against future objects, it has notable downsides. In particular passing the name at `VkCreateInfo` time does not allow renaming, prevents late binding of naming information, and

does not allow naming of implicitly created objects such as queues and swapchain images.

2) Should the command annotation functions `vkCmdDebugMarkerBeginEXT` and `vkCmdDebugMarkerEndEXT` support the ability to specify a color?

**RESOLVED:** Yes. The functions have been expanded to take an optional color which can be used at will by implementations consuming the command buffer annotations in their visualisation.

3) Should the functions added in this extension accept an extensible structure as their parameter for a more flexible API, as opposed to direct function parameters? If so, which functions?

**RESOLVED:** Yes. All functions have been modified to take a structure type with extensible `pNext` pointer, to allow future extensions to add additional annotation information in the same commands.

## Version History

- Revision 1, 2016-02-24 (Baldur Karlsson)
  - Initial draft, based on LunarG marker spec
- Revision 2, 2016-02-26 (Baldur Karlsson)
  - Renamed Dbg to DebugMarker in function names
  - Allow markers in secondary command buffers under certain circumstances
  - Minor language tweaks and edits
- Revision 3, 2016-04-23 (Baldur Karlsson)
  - Reorganise spec layout to closer match desired organisation
  - Added optional color to markers (both regions and inserted labels)
  - Changed functions to take extensible structs instead of direct function parameters
- Revision 4, 2017-01-31 (Baldur Karlsson)
  - Added explicit dependency on VK\_EXT\_debug\_report
  - Moved definition of `VkDebugReportObjectTypeEXT` to debug report chapter.
  - Fixed typo in dates in revision history

## VK\_EXT\_debug\_report

### Name String

`VK_EXT_debug_report`

### Extension Type

Instance extension

### Registered Extension Number

12

### Revision

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Deprecation state

- *Deprecated* by [VK\\_EXT\\_debug\\_utils](#) extension

## Contact

- Courtney Goeltzenleuchter [@courtney-g](#)

## Last Modified Date

2017-09-12

## IP Status

No known IP claims.

## Contributors

- Courtney Goeltzenleuchter, LunarG
- Dan Ginsburg, Valve
- Jon Ashburn, LunarG
- Mark Lobodzinski, LunarG

Due to the nature of the Vulkan interface, there is very little error information available to the developer and application. By enabling optional validation layers and using the [VK\\_EXT\\_debug\\_report](#) extension, developers **can** obtain much more detailed feedback on the application's use of Vulkan. This extension defines a way for layers and the implementation to call back to the application for events of interest to the application.

## New Object Types

- [VkDebugReportCallbackEXT](#)

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_DEBUG\\_REPORT\\_CALLBACK\\_CREATE\\_INFO\\_EXT](#)
- Extending [VkResult](#):
  - [VK\\_ERROR\\_VALIDATION\\_FAILED\\_EXT](#)

## New Enums

- [VkDebugReportFlagBitsEXT](#)
- [VkDebugReportObjectTypeEXT](#)

## New Structures

- [VkDebugReportCallbackCreateInfoEXT](#)

## New Functions

- [vkCreateDebugReportCallbackEXT](#)
- [vkDestroyDebugReportCallbackEXT](#)
- [vkDebugReportMessageEXT](#)

## New Function Pointers

- [PFN\\_vkDebugReportCallbackEXT](#)

## Examples

`VK_EXT_debug_report` allows an application to register multiple callbacks with the validation layers. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. An application **can** register callbacks even when no validation layers are enabled, but they will only be called for loader and, if implemented, driver events.

To capture events that occur while creating or destroying an instance an application **can** link a `VkDebugReportCallbackCreateInfoEXT` structure to the `pNext` element of the `VkInstanceCreateInfo` structure given to `vkCreateInstance`. This callback is only valid for the duration of the `vkCreateInstance` and the `vkDestroyInstance` call. Use `vkCreateDebugReportCallbackEXT` to create persistent callback objects.

Example uses: Create three callback objects. One will log errors and warnings to the debug console using Windows `OutputDebugString`. The second will cause the debugger to break at that callback when an error happens and the third will log warnings to stdout.

```

VkResult res;
VkDebugReportCallbackEXT cb1, cb2, cb3;

VkDebugReportCallbackCreateInfoEXT callback1 = {
    VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT, // sType
    NULL, // pNext
    VK_DEBUG_REPORT_ERROR_BIT_EXT | // flags
    VK_DEBUG_REPORT_WARNING_BIT_EXT,
    myOutputDebugString, // pfnCallback
    NULL // pUserData
};
res = vkCreateDebugReportCallbackEXT(instance, &callback1, &cb1);
if (res != VK_SUCCESS)
    /* Do error handling for VK_ERROR_OUT_OF_MEMORY */

callback.flags = VK_DEBUG_REPORT_ERROR_BIT_EXT;
callback.pfnCallback = myDebugBreak;
callback.pUserData = NULL;
res = vkCreateDebugReportCallbackEXT(instance, &callback, &cb2);
if (res != VK_SUCCESS)
    /* Do error handling for VK_ERROR_OUT_OF_MEMORY */

VkDebugReportCallbackCreateInfoEXT callback3 = {
    VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT, // sType
    NULL, // pNext
    VK_DEBUG_REPORT_WARNING_BIT_EXT, // flags
    mystdOutLogger, // pfnCallback
    NULL // pUserData
};
res = vkCreateDebugReportCallbackEXT(instance, &callback3, &cb3);
if (res != VK_SUCCESS)
    /* Do error handling for VK_ERROR_OUT_OF_MEMORY */

...

/* remove callbacks when cleaning up */
vkDestroyDebugReportCallbackEXT(instance, cb1);
vkDestroyDebugReportCallbackEXT(instance, cb2);
vkDestroyDebugReportCallbackEXT(instance, cb3);

```

#### Note

**i** In the initial release of the `VK_EXT_debug_report` extension, the token `VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT` was used. Starting in version 2 of the extension branch, `VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT` is used instead for consistency with Vulkan naming rules. The older enum is still available for backwards compatibility.

### Note



In the initial release of the `VK_EXT_debug_report` extension, the token `VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_EXT` was used. Starting in version 8 of the extension branch, `VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT_EXT` is used instead for consistency with Vulkan naming rules. The older enum is still available for backwards compatibility.

## Issues

1) What is the hierarchy / seriousness of the message flags? E.g. `ERROR > WARN > PERF_WARN ...`

**RESOLVED:** There is no specific hierarchy. Each bit is independent and should be checked via bitwise AND. For example:

```
if (localFlags & VK_DEBUG_REPORT_ERROR_BIT_EXT) {
    process error message
}
if (localFlags & VK_DEBUG_REPORT_DEBUG_BIT_EXT) {
    process debug message
}
```

The validation layers do use them in a hierarchical way (`ERROR > WARN > PERF, WARN > DEBUG > INFO`) and they (at least at the time of this writing) only set one bit at a time. But it is not a requirement of this extension.

It is possible that a layer may intercept and change, or augment the flags with extension values the application's debug report handler may not be familiar with, so it is important to treat each flag independently.

2) Should there be a VU requiring `VkDebugReportCallbackCreateInfoEXT::flags` to be non-zero?

**RESOLVED:** It may not be very useful, but we do not need VU statement requiring the `VkDebugReportCallbackCreateInfoEXT::msgFlags` at create-time to be non-zero. One can imagine that apps may prefer it as it allows them to set the mask as desired - including nothing - at runtime without having to check.

3) What is the difference between `VK_DEBUG_REPORT_DEBUG_BIT_EXT` and `VK_DEBUG_REPORT_INFORMATION_BIT_EXT`?

**RESOLVED:** `VK_DEBUG_REPORT_DEBUG_BIT_EXT` specifies information that could be useful debugging the Vulkan implementation itself.

## Version History

- Revision 1, 2015-05-20 (Courtney Goetzenleuchter)
  - Initial draft, based on LunarG KHR spec, other KHR specs
- Revision 2, 2016-02-16 (Courtney Goetzenleuchter)

- Update usage, documentation
- Revision 3, 2016-06-14 (Courtney Goetzenleuchter)
  - Update VK\_EXT\_DEBUG\_REPORT\_SPEC\_VERSION to indicate added support for vkCreateInstance and vkDestroyInstance
- Revision 4, 2016-12-08 (Mark Lobodzinski)
  - Added Display\_KHR, DisplayModeKHR extension objects
  - Added ObjectTable\_NVX, IndirectCommandsLayout\_NVX extension objects
  - Bumped spec revision
  - Retroactively added version history
- Revision 5, 2017-01-31 (Baldur Karlsson)
  - Moved definition of [VkDebugReportObjectTypeEXT](#) from debug marker chapter
- Revision 6, 2017-01-31 (Baldur Karlsson)
  - Added VK\_DEBUG\_REPORT\_OBJECT\_TYPE\_DESCRIPTOR\_UPDATE\_TEMPLATE\_KHR\_EXT
- Revision 7, 2017-04-20 (Courtney Goeltzenleuchter)
  - Clarify wording and address questions from developers.
- Revision 8, 2017-04-21 (Courtney Goeltzenleuchter)
  - Remove unused enum VkDebugReportErrorEXT
- Revision 9, 2017-09-12 (Tobias Hector)
  - Added interactions with Vulkan 1.1

## VK\_EXT\_validation\_flags

### Name String

[VK\\_EXT\\_validation\\_flags](#)

### Extension Type

Instance extension

### Registered Extension Number

62

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Deprecated* by [VK\\_EXT\\_validation\\_features](#) extension

### Contact

- Tobin Ehli [@tobine](#)

## Last Modified Date

2019-08-19

## IP Status

No known IP claims.

## Contributors

- Tobin Ehlis, Google
- Courtney Goeltzenleuchter, Google

This extension provides the `VkValidationFlagsEXT` struct that can be included in the `pNext` chain of the `VkInstanceCreateInfo` structure passed as the `pCreateInfo` parameter of `vkCreateInstance`. The structure contains an array of `VkValidationCheckEXT` values that will be disabled by the validation layers.

## Deprecation by VK\_EXT\_validation\_features

Functionality in this extension is subsumed into the `VK_EXT_validation_features` extension.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_VALIDATION_FLAGS_EXT`

## New Enums

- `VkValidationCheckEXT`

## New Structures

- `VkValidationFlagsEXT`

## New Functions

None.

## Issues

None.

## Version History

- Revision 2, 2019-08-19 (Mark Lobodzinski)
  - Marked as deprecated
- Revision 1, 2016-08-26 (Courtney Goeltzenleuchter)
  - Initial draft

## **VK\_AMD\_draw\_indirect\_count**

### **Name String**

`VK_AMD_draw_indirect_count`

### **Extension Type**

Device extension

### **Registered Extension Number**

34

### **Revision**

2

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Deprecation state**

- *Promoted* to [VK\\_KHR\\_draw间接计数扩展](#) extension

### **Contact**

- Daniel Rakos [@drakos-amd](#)

### **Last Modified Date**

2016-08-23

### **IP Status**

No known IP claims.

### **Contributors**

- Matthaeus G. Chajdas, AMD
- Derrick Owens, AMD
- Graham Sellers, AMD
- Daniel Rakos, AMD
- Dominik Witczak, AMD

This extension allows an application to source the number of draw calls for indirect draw calls from a buffer. This enables applications to generate arbitrary amounts of draw commands and execute them without host intervention.

## **New Functions**

- [vkCmdDrawIndirectCountAMD](#)
- [vkCmdDrawIndexedIndirectCountAMD](#)

## Promotion to VK\_KHR\_draw\_indirect\_count

All functionality in this extension is included in [VK\\_KHR\\_draw\\_indirect\\_count](#), with the suffix changed to KHR. The original type, enum and command names are still available as aliases of the core functionality.

## Version History

- Revision 2, 2016-08-23 (Dominik Witczak)
  - Minor fixes
- Revision 1, 2016-07-21 (Matthaeus Chajdas)
  - Initial draft

## VK\_AMD\_gpu\_shader\_half\_float

### Name String

`VK_AMD_gpu_shader_half_float`

### Extension Type

Device extension

### Registered Extension Number

37

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Deprecated* by [VK\\_KHR\\_shader\\_float16\\_int8](#) extension

### Contact

- Dominik Witczak [Qdominikwitczakamd](mailto:@dominikwitczakamd)

### Last Modified Date

2019-04-11

### IP Status

No known IP claims.

### Contributors

- Daniel Rakos, AMD
- Dominik Witczak, AMD
- Donglin Wei, AMD

- Graham Sellers, AMD
- Qun Lin, AMD
- Rex Xu, AMD

## External Dependencies

- `SPV_AMD_gpu_shader_half_float`

This extension adds support for using half float variables in shaders.

## Deprecation by `VK_KHR_shader_float16_int8`

Functionality in this extension was included in `VK_KHR_shader_float16_int8` extension, when `slink::VkPhysicalDeviceShaderFloat16Int8FeaturesKHR::shaderFloat16` is enabled.

## Version History

- Revision 2, 2019-04-11 (Tobias Hector)
  - Marked as deprecated
- Revision 1, 2016-09-21 (Dominik Witczak)
  - Initial draft

## `VK_AMD_gpu_shader_int16`

### Name String

`VK_AMD_gpu_shader_int16`

### Extension Type

Device extension

### Registered Extension Number

133

### Revision

2

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Deprecated* by `VK_KHR_shader_float16_int8` extension

### Contact

- Qun Lin [Qlinqun](#)

### Last Modified Date

2019-04-11

## IP Status

No known IP claims.

## Interactions and External Dependencies

- Requires the [SPV\\_AMD\\_gpu\\_shader\\_int16](#) SPIR-V extension.

## Contributors

- Daniel Rakos, AMD
- Dominik Witczak, AMD
- Matthaeus G. Chajdas, AMD
- Rex Xu, AMD
- Timothy Lottes, AMD
- Zhi Cai, AMD

## External Dependencies

- [SPV\\_AMD\\_gpu\\_shader\\_int16](#)

This extension adds support for using 16-bit integer variables in shaders.

## Deprecation by VK\_KHR\_shader\_float16\_int8

Functionality in this extension was included in [VK\\_KHR\\_shader\\_float16\\_int8](#) extension, when [VkPhysicalDeviceFeatures::shaderInt16](#) and [slink::VkPhysicalDeviceShaderFloat16Int8FeaturesKHR::shaderFloat16](#) are enabled.

## Version History

- Revision 2, 2019-04-11 (Tobias Hector)
  - Marked as deprecated
- Revision 1, 2017-06-18 (Dominik Witczak)
  - First version

## VK\_AMD\_negative\_viewport\_height

### Name String

[VK\\_AMD\\_negative\\_viewport\\_height](#)

### Extension Type

Device extension

### Registered Extension Number

36

### Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Deprecation state

- *Obsoleted* by [VK\\_KHR\\_maintenance1](#) extension
  - Which in turn was *promoted* to [Vulkan 1.1](#)

## Contact

- Matthaeus G. Chajdas [Qanteru](#)

## Last Modified Date

2016-09-02

## IP Status

No known IP claims.

## Contributors

- Matthaeus G. Chajdas, AMD
- Graham Sellers, AMD
- Baldur Karlsson

This extension allows an application to specify a negative viewport height. The result is that the viewport transformation will flip along the y-axis.

- Allow negative height to be specified in the `VkViewport::height` field to perform y-inversion of the clip-space to framebuffer-space transform. This allows apps to avoid having to use `gl_Position.y = -gl_Position.y` in shaders also targeting other APIs.

## Obsoletion by VK\_KHR\_maintenance1 and Vulkan 1.1

Functionality in this extension is included in [VK\\_KHR\\_maintenance1](#) and subsequently Vulkan 1.1. Due to some slight behavioral differences, this extension **must** not be enabled alongside [VK\\_KHR\\_maintenance1](#), or in an instance created with version 1.1 or later requested in [VkApplicationInfo::apiVersion](#).

## Version History

- Revision 1, 2016-09-02 (Matthaeus Chajdas)
  - Initial draft

## VK\_NV\_dedicated\_allocation

### Name String

`VK_NV_dedicated_allocation`

### Extension Type

Device extension

## Registered Extension Number

27

## Revision

1

## Extension and Version Dependencies

- Requires Vulkan 1.0

## Deprecation state

- *Deprecated* by [VK\\_KHR\\_dedicated\\_allocation](#) extension
  - Which in turn was *promoted* to [Vulkan 1.1](#)

## Contact

- Jeff Bolz [@jeffbolz\\_nv](#)

## Last Modified Date

2016-05-31

## IP Status

No known IP claims.

## Contributors

- Jeff Bolz, NVIDIA

This extension allows device memory to be allocated for a particular buffer or image resource, which on some devices can significantly improve the performance of that resource. Normal device memory allocations must support memory aliasing and sparse binding, which could interfere with optimizations like framebuffer compression or efficient page table usage. This is important for render targets and very large resources, but need not (and probably should not) be used for smaller resources that can benefit from suballocation.

This extension adds a few small structures to resource creation and memory allocation: a new structure that flags whether an image/buffer will have a dedicated allocation, and a structure indicating the image or buffer that an allocation will be bound to.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_BUFFER_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV`

## New Enums

None.

## New Structures

- [VkDedicatedAllocationImageCreateInfoNV](#)
- [VkDedicatedAllocationBufferCreateInfoNV](#)
- [VkDedicatedAllocationMemoryAllocateInfoNV](#)

## New Functions

None.

## Issues

None.

## Examples

```
// Create an image with
// VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation
// set to VK_TRUE

VkDedicatedAllocationImageCreateInfoNV dedicatedImageInfo =
{
    VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV,           //
    sType,                                                               //
    NULL,                                                               //
    pNext,                                                               //
    VK_TRUE,                                                            //
    dedicatedAllocation
};

VkImageCreateInfo imageCreateInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,        // sType
    &dedicatedImageInfo,                      // pNext
    // Other members set as usual
};

VkImage image;
VkResult result = vkCreateImage(
    device,
    &imageCreateInfo,
    NULL,                                // pAllocator
    &image);

VkMemoryRequirements memoryRequirements;
```

```

vkGetImageMemoryRequirements(
    device,
    image,
    &memoryRequirements);

// Allocate memory with VkDedicatedAllocationMemoryAllocateInfoNV::image
// pointing to the image we are allocating the memory for

VkDedicatedAllocationMemoryAllocateInfoNV dedicatedInfo =
{
    VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV,           //
sType
    NULL,                                                               //
pNext
    image,                                                               //
image
    VK_NULL_HANDLE,                                                 //
buffer
};

VkMemoryAllocateInfo memoryAllocateInfo =
{
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO,          // sType
    &dedicatedInfo,                                // pNext
    memoryRequirements.size,                         // allocationSize
    FindMemoryTypeIndex(memoryRequirements.memoryTypeBits), // memoryTypeIndex
};

VkDeviceMemory memory;
vkAllocateMemory(
    device,
    &memoryAllocateInfo,
    NULL,                                         // pAllocator
    &memory);

// Bind the image to the memory

vkBindImageMemory(
    device,
    image,
    memory,
    0);

```

## Version History

- Revision 1, 2016-05-31 (Jeff Bolz)
  - Internal revisions

## **VK\_NV\_external\_memory**

### **Name String**

`VK_NV_external_memory`

### **Extension Type**

Device extension

### **Registered Extension Number**

57

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires [VK\\_NV\\_external\\_memory\\_capabilities](#)

### **Deprecation state**

- *Deprecated* by [VK\\_KHR\\_external\\_memory](#) extension
  - Which in turn was *promoted* to [Vulkan 1.1](#)

### **Contact**

- James Jones [@cubanismo](#)

### **Last Modified Date**

2016-08-19

### **IP Status**

No known IP claims.

### **Contributors**

- James Jones, NVIDIA
- Carsten Rohde, NVIDIA

Applications may wish to export memory to other Vulkan instances or other APIs, or import memory from other Vulkan instances or other APIs to enable Vulkan workloads to be split up across application module, process, or API boundaries. This extension enables applications to create exportable Vulkan memory objects such that the underlying resources can be referenced outside the Vulkan instance that created them.

## **New Object Types**

None.

## New Enum Constants

Extending [VkStructureType](#):

- [VK\\_STRUCTURE\\_TYPE\\_EXTERNAL\\_MEMORY\\_IMAGE\\_CREATE\\_INFO\\_NV](#)
- [VK\\_STRUCTURE\\_TYPE\\_EXPORT\\_MEMORY\\_ALLOCATE\\_INFO\\_NV](#)

## New Enums

None.

## New Structures

- Extending [VkImageCreateInfo](#):
  - [VkExternalMemoryImageCreateInfoNV](#)
- Extending [VkMemoryAllocateInfo](#)
  - [VkExportMemoryAllocateInfoNV](#)

## New Functions

None.

## Issues

1) If memory objects are shared between processes and APIs, is this considered aliasing according to the rules outlined in the [Memory Aliasing](#) section?

**RESOLVED:** Yes, but strict exceptions to the rules are added to allow some forms of aliasing in these cases. Further, other extensions may build upon these new aliasing rules to define specific support usage within Vulkan for imported native memory objects, or memory objects from other APIs.

2) Are new image layouts or metadata required to specify image layouts and layout transitions compatible with non-Vulkan APIs, or with other instances of the same Vulkan driver?

**RESOLVED:** No. Separate instances of the same Vulkan driver running on the same GPU should have identical internal layout semantics, so applications have the tools they need to ensure views of images are consistent between the two instances. Other APIs will fall into two categories: Those that are Vulkan compatible (a term to be defined by subsequent interoperability extensions), or Vulkan incompatible. When sharing images with Vulkan incompatible APIs, the Vulkan image must be transitioned to the [VK\\_IMAGE\\_LAYOUT\\_GENERAL](#) layout before handing it off to the external API.

Note this does not attempt to address cross-device transitions, nor transitions to engines on the same device which are not visible within the Vulkan API. Both of these are beyond the scope of this extension.

## Examples

```
// TODO: Write some sample code here.
```

## Version History

- Revision 1, 2016-08-19 (James Jones)
  - Initial draft

## VK\_NV\_external\_memory\_capabilities

### Name String

`VK_NV_external_memory_capabilities`

### Extension Type

Instance extension

### Registered Extension Number

56

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0

### Deprecation state

- *Deprecated* by [VK\\_KHR\\_external\\_memory\\_capabilities](#) extension
  - Which in turn was *promoted* to [Vulkan 1.1](#)

### Contact

- James Jones [Qcubanismo](#)

### Last Modified Date

2016-08-19

### IP Status

No known IP claims.

### Interactions and External Dependencies

- Interacts with Vulkan 1.1.
- Interacts with [VK\\_KHR\\_dedicated\\_allocation](#).
- Interacts with [VK\\_NV\\_dedicated\\_allocation](#).

### Contributors

- James Jones, NVIDIA

Applications may wish to import memory from the Direct 3D API, or export memory to other

Vulkan instances. This extension provides a set of capability queries that allow applications determine what types of win32 memory handles an implementation supports for a given set of use cases.

## New Object Types

None.

## New Enum Constants

None.

## New Enums

- [VkExternalMemoryHandleTypeFlagBitsNV](#)
- [VkExternalMemoryFeatureFlagBitsNV](#)

## New Structs

- [VkExternalImageFormatPropertiesNV](#)

## New Functions

- [vkGetPhysicalDeviceExternalImageFormatPropertiesNV](#)

## Issues

1) Why do so many external memory capabilities need to be queried on a per-memory-handle-type basis?

**RESOLVED:** This is because some handle types are based on OS-native objects that have far more limited capabilities than the very generic Vulkan memory objects. Not all memory handle types can name memory objects that support 3D images, for example. Some handle types cannot even support the deferred image and memory binding behavior of Vulkan and require specifying the image when allocating or importing the memory object.

2) Does the [VkExternalImageFormatPropertiesNV](#) struct need to include a list of memory type bits that support the given handle type?

**RESOLVED:** No. The memory types that do not support the handle types will simply be filtered out of the results returned by [vkGetImageMemoryRequirements](#) when a set of handle types was specified at image creation time.

3) Should the non-opaque handle types be moved to their own extension?

**RESOLVED:** Perhaps. However, defining the handle type bits does very little and does not require any platform-specific types on its own, and it is easier to maintain the bitmask values in a single extension for now. Presumably more handle types could be added by separate extensions though, and it would be mildly weird to have some platform-specific ones defined in the core spec and some

in extensions

## Version History

- Revision 1, 2016-08-19 (James Jones)
  - Initial version

## VK\_NV\_external\_memory\_win32

### Name String

`VK_NV_external_memory_win32`

### Extension Type

Device extension

### Registered Extension Number

58

### Revision

1

### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_NV\\_external\\_memory](#)

### Deprecation state

- *Deprecated* by [VK\\_KHR\\_external\\_memory\\_win32](#) extension

### Contact

- James Jones [@cubanismo](#)

### Last Modified Date

2016-08-19

### IP Status

No known IP claims.

### Contributors

- James Jones, NVIDIA
- Carsten Rohde, NVIDIA

Applications may wish to export memory to other Vulkan instances or other APIs, or import memory from other Vulkan instances or other APIs to enable Vulkan workloads to be split up across application module, process, or API boundaries. This extension enables win32 applications to export win32 handles from Vulkan memory objects such that the underlying resources can be referenced outside the Vulkan instance that created them, and import win32 handles created in the Direct3D API to Vulkan memory objects.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - `VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_NV`
  - `VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_NV`

## New Enums

None.

## New Structures

- Extending [VkMemoryAllocateInfo](#)
  - [VkImportMemoryWin32HandleInfoNV](#)
- Extends [VkMemoryAllocateInfo](#)
  - [VkExportMemoryWin32HandleInfoNV](#)

## New Functions

- [vkGetMemoryWin32HandleNV](#)

## Issues

1) If memory objects are shared between processes and APIs, is this considered aliasing according to the rules outlined in the [Memory Aliasing](#) section?

**RESOLVED:** Yes, but strict exceptions to the rules are added to allow some forms of aliasing in these cases. Further, other extensions may build upon these new aliasing rules to define specific support usage within Vulkan for imported native memory objects, or memory objects from other APIs.

2) Are new image layouts or metadata required to specify image layouts and layout transitions compatible with non-Vulkan APIs, or with other instances of the same Vulkan driver?

**RESOLVED:** No. Separate instances of the same Vulkan driver running on the same GPU should have identical internal layout semantics, so applications have the tools they need to ensure views of images are consistent between the two instances. Other APIs will fall into two categories: Those that are Vulkan compatible (a term to be defined by subsequent interoperability extensions), or Vulkan incompatible. When sharing images with Vulkan incompatible APIs, the Vulkan image must be transitioned to the `VK_IMAGE_LAYOUT_GENERAL` layout before handing it off to the external API.

Note this does not attempt to address cross-device transitions, nor transitions to engines on the same device which are not visible within the Vulkan API. Both of these are beyond the scope of this extension.

3) Do applications need to call `CloseHandle()` on the values returned from `vkGetMemoryWin32HandleNV` when `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV`?

**RESOLVED:** Yes, unless it is passed back in to another driver instance to import the object. A successful get call transfers ownership of the handle to the application, while an import transfers ownership to the associated driver. Destroying the memory object will not destroy the handle or the handle's reference to the underlying memory resource.

## Examples

```
//  
// Create an exportable memory object and export an external  
// handle from it.  
  
//  
  
// Pick an external format and handle type.  
static const VkFormat format = VK_FORMAT_R8G8B8A8_UNORM;  
static const VkExternalMemoryHandleTypeFlagsNV handleType =  
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV;  
  
extern VkPhysicalDevice physicalDevice;  
extern VkDevice device;  
  
VkPhysicalDeviceMemoryProperties memoryProperties;  
VkExternalImageFormatPropertiesNV properties;  
VkExternalMemoryImageCreateInfoNV externalMemoryImageCreateInfo;  
VkDedicatedAllocationImageCreateInfoNV dedicatedImageCreateInfo;  
VkImageCreateInfo imageCreateInfo;  
VkImage image;  
VkMemoryRequirements imageMemoryRequirements;  
uint32_t numMemoryTypes;  
uint32_t memoryType;  
VkExportMemoryAllocateInfoNV exportMemoryAllocateInfo;  
VkDedicatedAllocationMemoryAllocateInfoNV dedicatedAllocationInfo;  
VkMemoryAllocateInfo memoryAllocateInfo;  
VkDeviceMemory memory;  
VkResult result;  
HANDLE memoryHnd;  
  
// Figure out how many memory types the device supports  
vkGetPhysicalDeviceMemoryProperties(physicalDevice,  
                                    &memoryProperties);  
numMemoryTypes = memoryProperties.memoryTypeCount;  
  
// Check the external handle type capabilities for the chosen format  
// Exportable 2D image support with at least 1 mip level, 1 array  
// layer, and VK_SAMPLE_COUNT_1_BIT using optimal tiling and supporting  
// texturing and color rendering is required.  
result = vkGetPhysicalDeviceExternalImageFormatPropertiesNV(
```

```

physicalDevice,
format,
VK_IMAGE_TYPE_2D,
VK_IMAGE_TILING_OPTIMAL,
VK_IMAGE_USAGE_SAMPLED_BIT |
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT,
0,
handleType,
&properties);

if ((result != VK_SUCCESS) ||
!(properties.externalMemoryFeatures &
VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_NV)) {
abort();
}

// Set up the external memory image creation info
memset(&externalMemoryImageCreateInfo,
0, sizeof(externalMemoryImageCreateInfo));
externalMemoryImageCreateInfo.sType =
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV;
externalMemoryImageCreateInfo.handleTypes = handleType;
if (properties.externalMemoryFeatures &
VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV) {
memset(&dedicatedImageCreateInfo, 0, sizeof(dedicatedImageCreateInfo));
dedicatedImageCreateInfo.sType =
VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV;
dedicatedImageCreateInfo.dedicatedAllocation = VK_TRUE;
externalMemoryImageCreateInfo.pNext = &dedicatedImageCreateInfo;
}
// Set up the core image creation info
memset(&imageCreateInfo, 0, sizeof(imageCreateInfo));
imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageCreateInfo.pNext = &externalMemoryImageCreateInfo;
imageCreateInfo.format = format;
imageCreateInfo.extent.width = 64;
imageCreateInfo.extent.height = 64;
imageCreateInfo.extent.depth = 1;
imageCreateInfo.mipLevels = 1;
imageCreateInfo.arrayLayers = 1;
imageCreateInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageCreateInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
imageCreateInfo.usage = VK_IMAGE_USAGE_SAMPLED_BIT |
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
imageCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
imageCreateInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

vkCreateImage(device, &imageCreateInfo, NULL, &image);

vkGetImageMemoryRequirements(device,
image,

```

```

        &imageMemoryRequirements);

    // For simplicity, just pick the first compatible memory type.
    for (memoryType = 0; memoryType < numMemoryTypes; memoryType++) {
        if ((1 << memoryType) & imageMemoryRequirements.memoryTypeBits) {
            break;
        }
    }

    // At least one memory type must be supported given the prior external
    // handle capability check.
    assert(memoryType < numMemoryTypes);

    // Allocate the external memory object.
    memset(&exportMemoryAllocateInfo, 0, sizeof(exportMemoryAllocateInfo));
    exportMemoryAllocateInfo.sType =
        VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV;
    exportMemoryAllocateInfo.handleTypes = handleType;
    if (properties.externalMemoryFeatures &
        VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV) {
        memset(&dedicatedAllocationInfo, 0, sizeof(dedicatedAllocationInfo));
        dedicatedAllocationInfo.sType =
            VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV;
        dedicatedAllocationInfo.image = image;
        exportMemoryAllocateInfo.pNext = &dedicatedAllocationInfo;
    }
    memset(&memoryAllocateInfo, 0, sizeof(memoryAllocateInfo));
    memoryAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    memoryAllocateInfo.pNext = &exportMemoryAllocateInfo;
    memoryAllocateInfo.allocationSize = imageMemoryRequirements.size;
    memoryAllocateInfo.memoryTypeIndex = memoryType;

    vkAllocateMemory(device, &memoryAllocateInfo, NULL, &memory);

    if (!(properties.externalMemoryFeatures &
          VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV)) {
        vkBindImageMemory(device, image, memory, 0);
    }

    // Get the external memory opaque FD handle
    vkGetMemoryWin32HandleNV(device, memory, &memoryHnd);

```

## Version History

- Revision 1, 2016-08-11 (James Jones)
  - Initial draft

## **VK\_NV\_glsl\_shader**

### **Name String**

`VK_NV_glsl_shader`

### **Extension Type**

Device extension

### **Registered Extension Number**

13

### **Revision**

1

### **Extension and Version Dependencies**

- Requires Vulkan 1.0

### **Deprecation state**

- *Deprecated* without replacement

### **Contact**

- Piers Daniell [Opdaniell-nv](#)

### **Last Modified Date**

2016-02-14

### **IP Status**

No known IP claims.

### **Contributors**

- Piers Daniell, NVIDIA

This extension allows GLSL shaders written to the `GL_KHR_vulkan_glsl` extension specification to be used instead of SPIR-V. The implementation will automatically detect whether the shader is SPIR-V or GLSL, and compile it appropriately.

## **New Object Types**

## **New Enum Constants**

- Extending `VkResult`:
  - `VK_ERROR_INVALID_SHADER_NV`

## **New Enums**

## **New Structures**

## New Functions

## Issues

## Examples

### Example 1

Passing in GLSL code

```
char const vss[] =
    "#version 450 core\n"
    "layout(location = 0) in vec2 aVertex;\n"
    "layout(location = 1) in vec4 aColor;\n"
    "out vec4 vColor;\n"
    "void main()\n"
    "{\n        vColor = aColor;\n        gl_Position = vec4(aVertex, 0, 1);\n    }\n";
VkShaderModuleCreateInfo vertexShaderInfo = {
VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO };
vertexShaderInfo.codeSize = sizeof vss;
vertexShaderInfo.pCode = vss;
VkShaderModule vertexShader;
vkCreateShaderModule(device, &vertexShaderInfo, 0, &vertexShader);
```

## Version History

- Revision 1, 2016-02-14 (Piers Daniell)
  - Initial draft

## VK\_NV\_win32\_keyed\_mutex

### Name String

VK\_NV\_win32\_keyed\_mutex

### Extension Type

Device extension

### Registered Extension Number

59

### Revision

2

## Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires [VK\\_NV\\_external\\_memory\\_win32](#)

## Deprecation state

- *Promoted* to [VK\\_KHR\\_win32\\_keyed\\_mutex](#) extension

## Contact

- Carsten Rohde [Ocrohde](#)

## Last Modified Date

2016-08-19

## IP Status

No known IP claims.

## Contributors

- James Jones, NVIDIA
- Carsten Rohde, NVIDIA

Applications that wish to import Direct3D 11 memory objects into the Vulkan API may wish to use the native keyed mutex mechanism to synchronize access to the memory between Vulkan and Direct3D. This extension provides a way for an application to access the keyed mutex associated with an imported Vulkan memory object when submitting command buffers to a queue.

## New Object Types

None.

## New Enum Constants

- Extending [VkStructureType](#):
  - [VK\\_STRUCTURE\\_TYPE\\_WIN32\\_KEYED\\_MUTEX\\_ACQUIRE\\_RELEASE\\_INFO\\_NV](#)

## New Enums

None.

## New Structures

- Extending [VkSubmitInfo](#):
  - [VkWin32KeyedMutexAcquireReleaseInfoNV](#)

## New Functions

None.

## Issues

None.

## Examples

```
//  
// Import a memory object from Direct3D 11, and synchronize  
// access to it in Vulkan using keyed mutex objects.  
//  
  
extern VkPhysicalDevice physicalDevice;  
extern VkDevice device;  
extern HANDLE sharedNtHandle;  
  
static const VkFormat format = VK_FORMAT_R8G8B8A8_UNORM;  
static const VkExternalMemoryHandleTypeFlagsNV handleType =  
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_BIT_NV;  
  
VkPhysicalDeviceMemoryProperties memoryProperties;  
VkExternalImageFormatPropertiesNV properties;  
VkExternalMemoryImageCreateInfoNV externalMemoryImageCreateInfo;  
VkImageCreateInfo imageCreateInfo;  
VkImage image;  
VkMemoryRequirements imageMemoryRequirements;  
uint32_t numMemoryTypes;  
uint32_t memoryType;  
VkImportMemoryWin32HandleInfoNV importMemoryInfo;  
VkMemoryAllocateInfo memoryAllocateInfo;  
VkDeviceMemory mem;  
VkResult result;  
  
// Figure out how many memory types the device supports  
vkGetPhysicalDeviceMemoryProperties(physicalDevice,  
                                    &memoryProperties);  
numMemoryTypes = memoryProperties.memoryTypeCount;  
  
// Check the external handle type capabilities for the chosen format  
// Importable 2D image support with at least 1 mip level, 1 array  
// layer, and VK_SAMPLE_COUNT_1_BIT using optimal tiling and supporting  
// texturing and color rendering is required.  
result = vkGetPhysicalDeviceExternalImageFormatPropertiesNV(  
    physicalDevice,  
    format,  
    VK_IMAGE_TYPE_2D,  
    VK_IMAGE_TILING_OPTIMAL,  
    VK_IMAGE_USAGE_SAMPLED_BIT |  
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT,  
    0,  
    handleType,
```

```

    &properties);

    if ((result != VK_SUCCESS) ||
        !(properties.externalMemoryFeatures &
          VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_NV)) {
        abort();
    }

    // Set up the external memory image creation info
    memset(&externalMemoryImageCreateInfo,
           0, sizeof(externalMemoryImageCreateInfo));
    externalMemoryImageCreateInfo.sType =
        VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV;
    externalMemoryImageCreateInfo.handleTypes = handleType;
    // Set up the core image creation info
    memset(&imageCreateInfo, 0, sizeof(imageCreateInfo));
    imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageCreateInfo.pNext = &externalMemoryImageCreateInfo;
    imageCreateInfo.format = format;
    imageCreateInfo.extent.width = 64;
    imageCreateInfo.extent.height = 64;
    imageCreateInfo.extent.depth = 1;
    imageCreateInfo.mipLevels = 1;
    imageCreateInfo.arrayLayers = 1;
    imageCreateInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageCreateInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
    imageCreateInfo.usage = VK_IMAGE_USAGE_SAMPLED_BIT |
        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
    imageCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    imageCreateInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

    vkCreateImage(device, &imageCreateInfo, NULL, &image);
    vkGetImageMemoryRequirements(device,
                                image,
                                &imageMemoryRequirements);

    // For simplicity, just pick the first compatible memory type.
    for (memoryType = 0; memoryType < numMemoryTypes; memoryType++) {
        if ((1 << memoryType) & imageMemoryRequirements.memoryTypeBits) {
            break;
        }
    }

    // At least one memory type must be supported given the prior external
    // handle capability check.
    assert(memoryType < numMemoryTypes);

    // Allocate the external memory object.
    memset(&exportMemoryAllocateInfo, 0, sizeof(exportMemoryAllocateInfo));
    exportMemoryAllocateInfo.sType =
        VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV;

```

```

importMemoryInfo.handleTypes = handleType;
importMemoryInfo.handle = sharedNtHandle;

memset(&memoryAllocateInfo, 0, sizeof(memoryAllocateInfo));
memoryAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
memoryAllocateInfo.pNext = &exportMemoryAllocateInfo;
memoryAllocateInfo.allocationSize = imageMemoryRequirements.size;
memoryAllocateInfo.memoryTypeIndex = memoryType;

vkAllocateMemory(device, &memoryAllocateInfo, NULL, &mem);

vkBindImageMemory(device, image, mem, 0);

...

const uint64_t acquireKey = 1;
const uint32_t timeout = INFINITE;
const uint64_t releaseKey = 2;

VkWin32KeyedMutexAcquireReleaseInfoNV keyedMutex =
    { VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_NV };
keyedMutex.acquireCount = 1;
keyedMutex.pAcquireSyncs = &mem;
keyedMutex.pAcquireKeys = &acquireKey;
keyedMutex.pAcquireTimeoutMilliseconds = &timeout;
keyedMutex.releaseCount = 1;
keyedMutex.pReleaseSyncs = &mem;
keyedMutex.pReleaseKeys = &releaseKey;

VkSubmitInfo submit_info = { VK_STRUCTURE_TYPE_SUBMIT_INFO, &keyedMutex };
submit_info.commandBufferCount = 1;
submit_info.pCommandBuffers = &cmd_buf;
vkQueueSubmit(queue, 1, &submit_info, VK_NULL_HANDLE);

```

## Version History

- Revision 2, 2016-08-11 (James Jones)
  - Updated sample code based on the NV external memory extensions.
  - Renamed from NVX to NV extension.
  - Added Overview and Description sections.
  - Updated sample code to use the NV external memory extensions.
- Revision 1, 2016-06-14 (Carsten Rohde)
  - Initial draft.

# Appendix F: API Boilerplate

This appendix defines Vulkan API features that are infrastructure required for a complete functional description of Vulkan, but do not logically belong elsewhere in the Specification.

## Vulkan Header Files

Vulkan is defined as an API in the C99 language. Khronos provides a corresponding set of header files for applications using the API, which may be used in either C or C++ code. The interface descriptions in the specification are the same as the interfaces defined in these header files, and both are derived from the `vk.xml` XML API Registry, which is the canonical machine-readable description of the Vulkan API. The Registry, scripts used for processing it into various forms, and documentation of the registry schema are available as described at <https://www.khronos.org/registry/vulkan/#apiregistry>.

Language bindings for other languages can be defined using the information in the Specification and the Registry. Khronos does not provide any such bindings, but third-party developers have created some additional bindings.

### Vulkan Combined API Header `vulkan.h` (Informative)

Applications normally will include the header `vulkan.h`. In turn, `vulkan.h` always includes the following headers:

- `vk_platform.h`, defining platform-specific macros and headers.
- `vulkan_core.h`, defining APIs for the Vulkan core and all registered extensions *other* than window system-specific extensions.

In addition, specific preprocessor macros defined at the time `vulkan.h` is included cause header files for the corresponding `window system-specific extension interfaces` to be included.

### Vulkan Platform-Specific Header `vk_platform.h` (Informative)

Platform-specific macros and interfaces are defined in `vk_platform.h`. These macros are used to control platform-dependent behavior, and their exact definitions are under the control of specific platforms and Vulkan implementations.

#### Platform-Specific Calling Conventions

On many platforms the following macros are empty strings, causing platform- and compiler-specific default calling conventions to be used.

`VKAPI_ATTR` is a macro placed before the return type in Vulkan API function declarations. This macro controls calling conventions for C++11 and GCC/Clang-style compilers.

`VKAPI_CALL` is a macro placed after the return type in Vulkan API function declarations. This macro controls calling conventions for MSVC-style compilers.

`VKAPI_PTR` is a macro placed between the '`(`' and '`*`' in Vulkan API function pointer declarations. This macro also controls calling conventions, and typically has the same definition as `VKAPI_ATTR` or `VKAPI_CALL`, depending on the compiler.

With these macros, a Vulkan function declaration takes the form of:

```
VKAPI_ATTR <return_type> VKAPI_CALL <command_name>(<command_parameters>);
```

Additionally, a Vulkan function pointer type declaration takes the form of:

```
typedef <return_type> (VKAPI_PTR *PFN_<command_name>)(<command_parameters>);
```

## Platform-Specific Header Control

If the `VK_NO_STDINT_H` macro is defined by the application at compile time, extended integer types used by the Vulkan API, such as `uint8_t`, must also be defined by the application. Otherwise, the Vulkan headers will not compile. If `VK_NO_STDINT_H` is not defined, the system `<stdint.h>` is used to define these types. There is a fallback path when Microsoft Visual Studio version 2008 and earlier versions are detected at compile time.

## Vulkan Core API Header `vulkan_core.h`

Applications that do not make use of window system-specific extensions may simply include `vulkan_core.h` instead of `vulkan.h`, although there is usually no reason to do so. In addition to the Vulkan API, `vulkan_core.h` also defines a small number of C preprocessor macros that are described below.

### Vulkan Header File Version Number

`VK_HEADER_VERSION` is the version number of the `vulkan_core.h` header. This value is kept synchronized with the patch version of the released Specification.

```
// Version of this file  
#define VK_HEADER_VERSION 129
```

`VK_API_VERSION` is now commented out of `vulkan_core.h` and cannot be used.

```
// DEPRECATED: This define has been removed. Specific version defines (e.g.  
VK_API_VERSION_1_0), or the VK_MAKE_VERSION macro, should be used instead.  
//#define VK_API_VERSION VK_MAKE_VERSION(1, 0, 0) // Patch version should always be  
set to 0
```

## Vulkan Handle Macros

`VK_DEFINE_HANDLE` defines a [dispatchable handle](#) type.

```
#define VK_DEFINE_HANDLE(object) typedef struct object##_T* object;
```

- **object** is the name of the resulting C type.

The only dispatchable handle types are those related to device and instance management, such as [VkDevice](#).

[VK\\_DEFINE\\_NON\\_DISPATCHABLE\\_HANDLE](#) defines a [non-dispatchable handle](#) type.

```
#if !defined(VK_DEFINE_NON_DISPATCHABLE_HANDLE)
#if defined(__LP64__) || defined(_WIN64) || (defined(__x86_64__) && !defined(
__ILP32__)) || defined(_M_X64) || defined(__ia64) || defined(_M_IA64) || defined(
__aarch64__) || defined(__powerpc64__)
    #define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object) typedef struct object##_T
*object;
#else
    #define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object) typedef uint64_t object;
#endif
#endif
```

- **object** is the name of the resulting C type.

Most Vulkan handle types, such as [VkBuffer](#), are non-dispatchable.

*Note*

The [vulkan\\_core.h](#) header allows the [VK\\_DEFINE\\_NON\\_DISPATCHABLE\\_HANDLE](#) definition to be overridden by the application. If [VK\\_DEFINE\\_NON\\_DISPATCHABLE\\_HANDLE](#) is already defined when [vulkan\\_core.h](#) is compiled, the default definition is skipped. This allows the application to define a binary-compatible custom handle which **may** provide more type-safety or other features needed by the application. Applications **must** not define handles in a way that is not binary compatible - where binary compatibility is platform dependent.

```
#define VK_NULL_HANDLE 0
```

## Window System-Specific Header Control (Informative)

To use a Vulkan extension supporting a platform-specific window system, header files for that window systems **must** be included at compile time, or platform-specific types **must** be forward-declared. The Vulkan header files cannot determine whether or not an external header is available at compile time, so platform-specific extensions are provided in separate headers from the core API and platform-independent extensions, allowing applications to decide which ones should be

defined and how the external headers are included.

Extensions dependent on particular sets of platform headers, or that forward-declare platform-specific types, are declared in a header named for that platform. Before including these platform-specific Vulkan headers, applications **must** include both `vulkan_core.h` and any external native headers the platform extensions depend on.

As a convenience for applications that do not need the flexibility of separate platform-specific Vulkan headers, `vulkan.h` includes `vulkan_core.h`, and then conditionally includes platform-specific Vulkan headers and the external headers they depend on. Applications control which platform-specific headers are included by #defining macros before including `vulkan.h`.

The correspondence between platform-specific extensions, external headers they require, the platform-specific header which declares them, and the preprocessor macros which enable inclusion by `vulkan.h` are shown in the [following table](#).

*Table 84. Window System Extensions and Headers*

Extension Name	Window System Name	Platform-specific Header	Required External Headers	Controlling <code>vulkan.h</code> Macro
<code>VK_KHR_android_surface</code>	Android	<code>vulkan_android.h</code>	None	<code>VK_USE_PLATFORM_ANDROID_KHR</code>
<code>VK_KHR_wayland_surface</code>	Wayland	<code>vulkan_wayland.h</code>	<code>&lt;wayland-client.h&gt;</code>	<code>VK_USE_PLATFORM_WAYLAND_KHR</code>
<code>VK_KHR_win32_surface</code> , <code>VK_KHR_external_memory_win32</code> , <code>VK_KHR_win32_keyed_mutex</code> , <code>VK_KHR_external_semaphore_win32</code> , <code>VK_KHR_external_fence_win32</code> , <code>VK_NV_external_memory_win32</code> , <code>VK_NV_win32_keyed_mutex</code>	Microsoft Windows	<code>vulkan_win32.h</code>	<code>&lt;windows.h&gt;</code>	<code>VK_USE_PLATFORM_WIN32_KHR</code>
<code>VK_KHR_xcb_surface</code>	X11 Xcb	<code>vulkan_xcb.h</code>	<code>&lt;xcb xcb.h&gt;</code>	<code>VK_USE_PLATFORM_XCB_KHR</code>
<code>VK_KHR_xlib_surface</code>	X11 Xlib	<code>vulkan_xlib.h</code>	<code>&lt;X11/Xlib.h&gt;</code>	<code>VK_USE_PLATFORM_XLIB_KHR</code>
<code>VK_EXT_acquire_xlib_display</code>	X11 XRandR	<code>vulkan_xlib_xrandr.h</code>	<code>&lt;X11/Xlib.h&gt;</code> , <code>&lt;X11/extensions/Xrandr.h&gt;</code>	<code>VK_USE_PLATFORM_XLIB_XRANDR_EXT</code>
<code>VK_GGP_stream_descriptor_surface</code> , <code>VK_GGP_frame_token</code>	Google Games Platform	<code>vulkan_ggp.h</code>	<code>&lt;ggp_c/vulkan_types.h&gt;</code>	<code>VK_USE_PLATFORM_GGP</code>
<code>VK_MVK_ios_surface</code>	iOS	<code>vulkan_ios.h</code>	None	<code>VK_USE_PLATFORM_IOS_MVK</code>
<code>VK_MVK_macos_surface</code>	macOS	<code>vulkan_macos.h</code>	None	<code>VK_USE_PLATFORM_MACOS_MVK</code>

Extension Name	Window System Name	Platform-specific Header	Required External Headers	Controlling <code>vulkan.h</code> Macro
<code>VK_NN_vi_surface</code>	VI	<code>vulkan_vi.h</code>	None	<code>VK_USE_PLATFORM_VI_NN</code>
<code>VK_FUCHSIA_imagepipe_surface</code>	Fuchsia	<code>vulkan_fuchsia.h</code>	<code>&lt;zircon/types.h&gt;</code>	<code>VK_USE_PLATFORM_FUCHSIA</code>
<code>VK_EXT_metal_surface</code>	Metal on CoreAnimation	<code>vulkan_metal.h</code>	None	<code>VK_USE_PLATFORM_METAL_EXT</code>

*Note*



This section describes the purpose of the headers independently of the specific underlying functionality of the window system extensions themselves. Each extension name will only link to a description of that extension when viewing a specification built with that extension included.

# Appendix G: Invariance

The Vulkan specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different Vulkan implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

## Repeatability

The obvious and most fundamental case is repeated issuance of a series of Vulkan commands. For any given Vulkan and framebuffer state vector, and for any Vulkan command, the resulting Vulkan and framebuffer state **must** be identical whenever the command is executed on that initial Vulkan and framebuffer state. This repeatability requirement does not apply when using shaders containing side effects (image and buffer variable stores and atomic operations), because these memory operations are not guaranteed to be processed in a defined order.

The repeatability requirement does not apply for rendering done using a graphics pipeline that uses `VK_RASTERIZATION_ORDER_RELAXED_AMD`.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence **may** result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

## Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different Vulkan mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- “Erasing” a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

## Invariance Rules

For a given Vulkan device:

**Rule 1** *For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the resulting Vulkan and framebuffer state **must** be identical each time the command is executed on that initial Vulkan and framebuffer state.*

**Rule 2** Changes to the following state values have no side effects (the use of any other state value is not affected by the change):

**Required:**

- Color and depth/stencil attachment contents
- Scissor parameters (other than enable)
- Write masks (color, depth, stencil)
- Clear values (color, depth, stencil)

**Strongly suggested:**

- Stencil parameters (other than enable)
- Depth test parameters (other than enable)
- Blend parameters (other than enable)
- Logical operation parameters (other than enable)

**Corollary 1** Fragment generation is invariant with respect to the state values listed in Rule 2.

**Rule 3** The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it.

**Corollary 2** Images rendered into different color attachments of the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.

**Rule 4** Identical pipelines will produce the same result when run multiple times with the same input. The wording “Identical pipelines” means [VkPipeline](#) objects that have been created with identical SPIR-V binaries and identical state, which are then used by commands executed using the same Vulkan state vector. Invariance is relaxed for shaders with side effects, such as performing stores or atomics.

**Rule 5** All fragment shaders that either conditionally or unconditionally assign [FragCoord.z](#) to [FragDepth](#) are depth-invariant with respect to each other, for those fragments where the assignment to [FragDepth](#) actually is done.

If a sequence of Vulkan commands specifies primitives to be rendered with shaders containing side effects (image and buffer variable stores and atomic operations), invariance rules are relaxed. In particular, rule 1, corollary 2, and rule 4 do not apply in the presence of shader side effects.

The following weaker versions of rules 1 and 4 apply to Vulkan commands involving shader side effects:

**Rule 6** For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the contents of any framebuffer state not directly or indirectly affected by results of shader image or buffer variable stores or atomic operations **must** be identical each time the command is executed on that initial Vulkan and framebuffer state.

**Rule 7** Identical pipelines will produce the same result when run multiple times with the same input

as long as:

- shader invocations do not use image atomic operations;
- no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and
- no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.

**Note**

The OpenGL spec has the following invariance rule: Consider a primitive  $p'$  obtained by translating a primitive  $p$  through an offset  $(x, y)$  in window coordinates, where  $x$  and  $y$  are integers. As long as neither  $p'$  nor  $p$  is clipped, it **must** be the case that each fragment  $f'$  produced from  $p'$  is identical to a corresponding fragment  $f$  from  $p$  except that the center of  $f'$  is offset by  $(x, y)$  from the center of  $f$ .



This rule does not apply to Vulkan and is an intentional difference from OpenGL.

When any sequence of Vulkan commands triggers shader invocations that perform image stores or atomic operations, and subsequent Vulkan commands read the memory written by those shader invocations, these operations **must** be explicitly synchronized.

## Tessellation Invariance

When using a pipeline containing tessellation evaluation shaders, the fixed-function tessellation primitive generator consumes the input patch specified by an application and emits a new set of primitives. The following invariance rules are intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding “cracks” caused by minor differences in the positions of vertices along shared edges.

**Rule 1** *When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line, or triangle primitives as long as the pipeline used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode decorations. Two sets of primitives are considered identical if and only if they contain the same number and type of primitives and the generated tessellation coordinates for the vertex numbered  $m$  of the primitive numbered  $n$  are identical for all values of  $m$  and  $n$ .*

**Rule 2** *The set of vertices generated along the outer edge of the subdivided primitive in triangle and quad tessellation, and the tessellation coordinates of each, depends only on the corresponding outer tessellation level and the spacing decorations in the tessellation shaders of the pipeline.*

**Rule 3** *The set of vertices generated when subdividing any outer primitive edge is always symmetric. For triangle tessellation, if the subdivision generates a vertex with tessellation coordinates of the form  $(0, x, 1-x)$ ,  $(x, 0, 1-x)$ , or  $(x, 1-x, 0)$ , it will also generate a vertex with coordinates of exactly  $(0, 1-x, x)$ ,  $(1-x, 0, x)$ , or  $(1-x, x, 0)$ , respectively. For quad tessellation, if the subdivision generates a vertex with*

*coordinates of  $(x, 0)$  or  $(0, x)$ , it will also generate a vertex with coordinates of exactly  $(1-x, 0)$  or  $(0, 1-x)$ , respectively. For isoline tessellation, if it generates vertices at  $(0, x)$  and  $(1, x)$  where  $x$  is not zero, it will also generate vertices at exactly  $(0, 1-x)$  and  $(1, 1-x)$ , respectively.*

**Rule 4** *The set of vertices generated when subdividing outer edges in triangular and quad tessellation **must** be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at  $(x, 1 - x, 0)$  and  $(1-x, x, 0)$  are generated when subdividing the  $w = 0$  edge in triangular tessellation, vertices **must** be generated at  $(x, 0, 1-x)$  and  $(1-x, 0, x)$  when subdividing an otherwise identical  $v = 0$  edge. For quad tessellation, if vertices at  $(x, 0)$  and  $(1-x, 0)$  are generated when subdividing the  $v = 0$  edge, vertices **must** be generated at  $(0, x)$  and  $(0, 1-x)$  when subdividing an otherwise identical  $u = 0$  edge.*

**Rule 5** *When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation **must** be identical except for vertex and triangle order. For each triangle  $n1$  produced by processing the first patch, there **must** be a triangle  $n2$  produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in  $n1$ .*

**Rule 6** *When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set of interior triangles generated for triangle and quad tessellation **must** be identical in all respects except for vertex and triangle order. For each interior triangle  $n1$  produced by processing the first patch, there **must** be a triangle  $n2$  produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in  $n1$ . A triangle produced by the tessellator is considered an interior triangle if none of its vertices lie on an outer edge of the subdivided primitive.*

**Rule 7** *For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing decorations.*

**Rule 8** *The value of all defined components of `TessCoord` will be in the range  $[0, 1]$ . Additionally, for any defined component  $x$  of `TessCoord`, the results of computing  $1.0-x$  in a tessellation evaluation shader will be exact. If any floating-point values in the range  $[0, 1]$  fail to satisfy this property, such values **must** not be used as tessellation coordinate components.*

# Glossary

The terms defined in this section are used consistently throughout this Specification and may be used with or without capitalization.

## Accessible (Descriptor Binding)

A descriptor binding is accessible to a shader stage if that stage is included in the `stageFlags` of the descriptor binding. Descriptors using that binding **can** only be used by stages in which they are accessible.

## Acquire Operation (Resource)

An operation that acquires ownership of an image subresource or buffer range.

## Active (Transform Feedback)

Transform feedback is made active after `vkCmdBeginTransformFeedbackEXT` executes and remains active until `vkCmdEndTransformFeedbackEXT` executes. While transform feedback is active, data written to variables in the output interface of the last vertex processing stage of the graphics pipeline are captured to the bound transform feedback buffers if those variables are decorated for transform feedback.

## Adjacent Vertex

A vertex in an adjacency primitive topology that is not part of a given primitive, but is accessible in geometry shaders.

## Advanced Blend Operation

Blending performed using one of the blend operation enums introduced by the `VK_EXT_blend_operation_advanced` extension. See [Advanced Blending Operations](#).

## Alias (API type/command)

An identical definition of another API type/command with the same behavior but a different name.

## Aliased Range (Memory)

A range of a device memory allocation that is bound to multiple resources simultaneously.

## Allocation Scope

An association of a host memory allocation to a parent object or command, where the allocation's lifetime ends before or at the same time as the parent object is freed or destroyed, or during the parent command.

## Aspect (Image)

An image **may** contain multiple kinds, or aspects, of data for each pixel, where each aspect is used in a particular way by the pipeline and **may** be stored differently or separately from other aspects. For example, the color components of an image format make up the color aspect of the image, and **may** be used as a framebuffer color attachment. Some operations, like depth testing, operate only on specific aspects of an image. Others operations, like image/buffer copies, only operate on one aspect at a time.

## **Attachment (Render Pass)**

A zero-based integer index name used in render pass creation to refer to a framebuffer attachment that is accessed by one or more subpasses. The index also refers to an attachment description which includes information about the properties of the image view that will later be attached.

## **Availability Operation**

An operation that causes the values generated by specified memory write accesses to become available for future access.

## **Available**

A state of values written to memory that allows them to be made visible.

## **Axis-aligned Bounding Box**

A box bounding a region in space defined by extents along each axis and thus representing a box where each edge is aligned to one of the major axes.

## **Back-Facing**

See Facingness.

## **Batch**

A single structure submitted to a queue as part of a [queue submission command](#), describing a set of queue operations to execute.

## **Backwards Compatibility**

A given version of the API is backwards compatible with an earlier version if an application, relying only on valid behavior and functionality defined by the earlier specification, is able to correctly run against each version without any modification. This assumes no active attempt by that application to not run when it detects a different version.

## **Binary Semaphore**

A semaphore with a boolean payload indicating whether the semaphore is signaled or unsignaled. Represented by a [VkSemaphore](#) object created with a semaphore type of [VK\\_SEMAPHORE\\_TYPE\\_BINARY\\_KHR](#).

## **Full Compatibility**

A given version of the API is fully compatible with another version if an application, relying only on valid behavior and functionality defined by either of those specifications, is able to correctly run against each version without any modification. This assumes no active attempt by that application to not run when it detects a different version.

## **Binding (Memory)**

An association established between a range of a resource object and a range of a memory object. These associations determine the memory locations affected by operations performed on elements of a resource object. Memory bindings are established using the [vkBindBufferMemory](#) command for non-sparse buffer objects, using the [vkBindImageMemory](#) command for non-sparse image objects, and using the [vkQueueBindSparse](#) command for sparse resources.

## **Blend Constant**

Four floating point (RGBA) values used as an input to blending.

## **Blending**

Arithmetic operations between a fragment color value and a value in a color attachment that produce a final color value to be written to the attachment.

## **Buffer**

A resource that represents a linear array of data in device memory. Represented by a [VkBuffer](#) object.

## **Buffer Device Address**

A 64-bit value used in a shader to access buffer memory through the [PhysicalStorageBuffer](#) storage class.

## **Buffer View**

An object that represents a range of a specific buffer, and state that controls how the contents are interpreted. Represented by a [VkBufferView](#) object.

## **Built-In Variable**

A variable decorated in a shader, where the decoration makes the variable take values provided by the execution environment or values that are generated by fixed-function pipeline stages.

## **Built-In Interface Block**

A block defined in a shader that contains only variables decorated with built-in decorations, and is used to match against other shader stages.

## **Clip Coordinates**

The homogeneous coordinate space that vertex positions ([Position](#) decoration) are written in by vertex processing stages.

## **Clip Distance**

A built-in output from vertex processing stages that defines a clip half-space against which the primitive is clipped.

## **Clip Volume**

The intersection of the view volume with all clip half-spaces.

## **Color Attachment**

A subpass attachment point, or image view, that is the target of fragment color outputs and blending.

## **Color Fragment**

A unique color value within a pixel of a multisampled color image. The *fragment mask* will contain indices to the *color fragment*.

## **Color Renderable Format**

A [VkFormat](#) where [VK\\_FORMAT\\_FEATURE\\_COLOR\\_ATTACHMENT\\_BIT](#) is set in one of the following,

depending on the image's tiling:

- `VkImageFormatProperties::linearTilingFeatures`
- `VkImageFormatProperties::optimalTilingFeatures`
- `VkDrmFormatModifierPropertiesEXT::drmFormatModifierTilingFeatures`

## Color Sample Mask

A bitfield associated with a fragment, with one bit for each sample in the color attachment(s). Samples are considered to be covered based on the result of the Coverage Reduction stage. Uncovered samples do not write to color attachments.

## Combined Image Sampler

A descriptor type that includes both a sampled image and a sampler.

## Command Buffer

An object that records commands to be submitted to a queue. Represented by a `VkCommandBuffer` object.

## Command Pool

An object that command buffer memory is allocated from, and that owns that memory. Command pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a `VkCommandPool` object.

## Compatible Allocator

When allocators are compatible, allocations from each allocator **can** be freed by the other allocator.

## Compatible Image Formats

When formats are compatible, images created with one of the formats **can** have image views created from it using any of the compatible formats. Also see *Size-Compatible Image Formats*.

## Compatible Queues

Queues within a queue family. Compatible queues have identical properties.

## Complete Mipmap Chain

The entire set of miplevels that can be provided for an image, from the largest application specified miplevel size down to the *minimum miplevel size*. See [Image Miplevel Sizing](#).

## Component (Format)

A distinct part of a format. Depth, stencil, and color channels (e.g. R, G, B, A), are all separate components.

## Compressed Texel Block

An element of an image having a block-compressed format, comprising a rectangular block of texel values that are encoded as a single value in memory. Compressed texel blocks of a particular block-compressed format have a corresponding width, height, and depth that define the dimensions of these elements in units of texels, and a size in bytes of the encoding in

memory.

## Constant Integral Expressions

A SPIR-V constant instruction whose type is [OpTypeInt](#). See *Constant Instruction* in section 2.2.1 “Instructions” of the [Khronos SPIR-V Specification](#).

## Cooperative Matrix

A SPIR-V type where the storage for and computations performed on the matrix are spread across a set of invocations such as a subgroup.

## Corner-Sampled Image

A [VkImage](#) where unnormalized texel coordinates are centered on integer values instead of half-integer values. Specified by setting the `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` bit on [VkImageCreateInfo::flags](#) at image creation.

## Coverage

A bitfield associated with a fragment, where each bit is associated to a rasterization sample. Samples are initially considered to be covered based on the result of rasterization, and then coverage can subsequently be turned on or off by other fragment operations or the fragment shader. Uncovered samples do not write to framebuffer attachments.

## Cull Distance

A built-in output from vertex processing stages that defines a cull half-space where the primitive is rejected if all vertices have a negative value for the same cull distance.

## Cull Volume

The intersection of the view volume with all cull half-spaces.

## Decoration (SPIR-V)

Auxiliary information such as built-in variables, stream numbers, invariance, interpolation type, relaxed precision, etc., added to variables or structure-type members through decorations.

## Deprecated (feature)

A feature is deprecated if it is no longer recommended as the correct or best way to achieve its intended purpose.

## Depth/Stencil Attachment

A subpass attachment point, or image view, that is the target of depth and/or stencil test operations and writes.

## Depth/Stencil Format

A [VkFormat](#) that includes depth and/or stencil components.

## Depth/Stencil Image (or ImageView)

A [VkImage](#) (or [VkImageView](#)) with a depth/stencil format.

## Depth/Stencil Resolve Attachment

A subpass attachment point, or image view, that is the target of a multisample resolve operation

from the corresponding depth/stencil attachment at the end of the subpass.

## Derivative Group

A set of fragment or compute shader invocations that cooperate to compute derivatives, including implicit derivatives for sampled image operations.

## Descriptor

Information about a resource or resource view written into a descriptor set that is used to access the resource or view from a shader.

## Descriptor Binding

An entry in a descriptor set layout corresponding to zero or more descriptors of a single descriptor type in a set. Defined by a [VkDescriptorSetLayoutBinding](#) structure.

## Descriptor Pool

An object that descriptor sets are allocated from, and that owns the storage of those descriptor sets. Descriptor pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a [VkDescriptorPool](#) object.

## Descriptor Set

An object that resource descriptors are written into via the API, and that **can** be bound to a command buffer such that the descriptors contained within it **can** be accessed from shaders. Represented by a [VkDescriptorSet](#) object.

## Descriptor Set Layout

An object that defines the set of resources (types and counts) and their relative arrangement (in the binding namespace) within a descriptor set. Used when allocating descriptor sets and when creating pipeline layouts. Represented by a [VkDescriptorSetLayout](#) object.

## Device

The processor(s) and execution environment that perform tasks requested by the application via the Vulkan API.

## Device Group

A set of physical devices that support accessing each other's memory and recording a single command buffer that **can** be executed on all the physical devices.

## Device Index

A zero-based integer that identifies one physical device from a logical device. A device index is valid if it is less than the number of physical devices in the logical device.

## Device Mask

A bitmask where each bit represents one device index. A device mask value is valid if every bit that is set in the mask is at a bit position that is less than the number of physical devices in the logical device.

## Device Memory

Memory accessible to the device. Represented by a [VkDeviceMemory](#) object.

## Device-Level Command

Any command that is dispatched from a logical device, or from a child object of a logical device.

## Device-Level Functionality

All device-level commands and objects, and their structures, enumerated types, and enumerants.

## Device-Level Object

Logical device objects and their child objects. For example, [VkDevice](#), [VkQueue](#), and [VkCommandBuffer](#) objects are device-level objects.

## Device-Local Memory

Memory that is connected to the device, and **may** be more performant for device access than host-local memory.

## Direct Drawing Commands

*Drawing commands* that take all their parameters as direct arguments to the command (and not sourced via structures in buffer memory as the *indirect drawing commands*). Includes [vkCmdDrawMeshTasksNV](#), [vkCmdDraw](#), and [vkCmdDrawIndexed](#).

## Disjoint

*Disjoint planes* are *image planes* to which memory is bound independently.

A *disjoint image* consists of multiple *disjoint planes*, and is created with the [VK\\_IMAGE\\_CREATE\\_DISJOINT\\_BIT](#) bit set.

## Dispatchable Handle

A handle of a pointer handle type which **may** be used by layers as part of intercepting API commands. The first argument to each Vulkan command is a dispatchable handle type.

## Dispatching Commands

Commands that provoke work using a compute pipeline. Includes [vkCmdDispatch](#) and [vkCmdDispatchIndirect](#).

## Drawing Commands

Commands that provoke work using a graphics pipeline. Includes [vkCmdDraw](#), [vkCmdDrawIndexed](#), [vkCmdDrawIndirectCountKHR](#), [vkCmdDrawIndexedIndirectCountKHR](#), [vkCmdDrawIndirectCountAMD](#), [vkCmdDrawIndexedIndirectCountAMD](#), [vkCmdDrawMeshTasksNV](#), [vkCmdDrawMeshTasksIndirectNV](#), [vkCmdDrawMeshTasksIndirectCountNV](#), [vkCmdDrawIndirect](#), and [vkCmdDrawIndexedIndirect](#).

## Duration (Command)

The *duration* of a Vulkan command refers to the interval between calling the command and its return to the caller.

## Dynamic Storage Buffer

A storage buffer whose offset is specified each time the storage buffer is bound to a command buffer via a descriptor set.

## **Dynamic Uniform Buffer**

A uniform buffer whose offset is specified each time the uniform buffer is bound to a command buffer via a descriptor set.

## **Dynamically Uniform**

See *Dynamically Uniform* in section 2.2 “Terms” of the Khronos SPIR-V Specification.

## **Element**

Arrays are composed of multiple elements, where each element exists at a unique index within that array. Used primarily to describe data passed to or returned from the Vulkan API.

## **Explicitly-Enabled Layer**

A layer enabled by the application by adding it to the enabled layer list in [vkCreateInstance](#) or [vkCreateDevice](#).

## **Event**

A synchronization primitive that is signaled when execution of previous commands complete through a specified set of pipeline stages. Events can be waited on by the device and polled by the host. Represented by a [VkEvent](#) object.

## **Executable State (Command Buffer)**

A command buffer that has ended recording commands and **can** be executed. See also Initial State and Recording State.

## **Execution Dependency**

A dependency that guarantees that certain pipeline stages’ work for a first set of commands has completed execution before certain pipeline stages’ work for a second set of commands begins execution. This is accomplished via pipeline barriers, subpass dependencies, events, or implicit ordering operations.

## **Execution Dependency Chain**

A sequence of execution dependencies that transitively act as a single execution dependency.

## **Explicit chroma reconstruction**

An implementation of sampler Y’C<sub>B</sub>C<sub>R</sub> conversion which reconstructs reduced-resolution chroma samples to luma resolution and then separately performs texture sample interpolation. This is distinct from an implicit implementation, which incorporates chroma sample reconstruction into texture sample interpolation.

## **Extension Scope**

The set of objects and commands that **can** be affected by an extension. Extensions are either device scope or instance scope.

## **External Handle**

A resource handle which has meaning outside of a specific Vulkan device or its parent instance. External handles **may** be used to share resources between multiple Vulkan devices in different instances, or between Vulkan and other APIs. Some external handle types correspond to platform-defined handles, in which case the resource **may** outlive any particular Vulkan device

or instance and **may** be transferred between processes, or otherwise manipulated via functionality defined by the platform for that handle type.

## External synchronization

A type of synchronization **required** of the application, where parameters defined to be externally synchronized **must** not be used simultaneously in multiple threads.

## Facingness (Polygon)

A classification of a polygon as either front-facing or back-facing, depending on the orientation (winding order) of its vertices.

## Facingness (Fragment)

A fragment is either front-facing or back-facing, depending on the primitive it was generated from. If the primitive was a polygon (regardless of polygon mode), the fragment inherits the facingness of the polygon. All other fragments are front-facing.

## Fence

A synchronization primitive that is signaled when a set of batches or sparse binding operations complete execution on a queue. Fences **can** be waited on by the host. Represented by a [VkFence](#) object.

## Flat Shading

A property of a vertex attribute that causes the value from a single vertex (the provoking vertex) to be used for all vertices in a primitive, and for interpolation of that attribute to return that single value unaltered.

## Fragment

A rectangular framebuffer region with associated data produced by [rasterization](#) and processed by [fragment operations](#) including the fragment shader.

## Fragment Area

The width and height, in pixels, of a fragment.

## Fragment Density

The ratio of fragments per framebuffer area in the x and y direction.

## Fragment Density Texel Size

The (w,h) framebuffer region in pixels that each texel in a fragment density map applies to.

## Fragment Input Attachment Interface

Variables with [UniformConstant](#) storage class and a decoration of [InputAttachmentIndex](#) that are statically used by a fragment shader's entry point, which receive values from input attachments.

## Fragment Mask

A lookup table that associates color samples with color fragment values.

## Fragment Output Interface

A fragment shader entry point's variables with [Output](#) storage class, which output to color and/or

depth/stencil attachments.

## Framebuffer

A collection of image views and a set of dimensions that, in conjunction with a render pass, define the inputs and outputs used by drawing commands. Represented by a [VkFramebuffer](#) object.

## Framebuffer Attachment

One of the image views used in a framebuffer.

## Framebuffer Coordinates

A coordinate system in which adjacent pixels' coordinates differ by 1 in x and/or y, with (0,0) in the upper left corner and pixel centers at half-integers.

## Framebuffer-Space

Operating with respect to framebuffer coordinates.

## Framebuffer-Local

A framebuffer-local dependency guarantees that only for a single framebuffer region, the first set of operations happens-before the second set of operations.

## Framebuffer-Global

A framebuffer-global dependency guarantees that for all framebuffer regions, the first set of operations happens-before the second set of operations.

## Framebuffer Region

A framebuffer region is a set of sample (x, y, layer, sample) coordinates that is a subset of the entire framebuffer.

## Front-Facing

See Facingness.

## Global Workgroup

A collection of local workgroups dispatched by a single dispatch command. In addition to the compute dispatch, a single mesh task draw command can also generate such a collection.

## Handle

An opaque integer or pointer value used to refer to a Vulkan object. Each object type has a unique handle type.

## Happen-after

A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of **A** and a destination of **B** enforces that **B** happens-after **A**. The inverse relation of happens-before.

## Happen-before

A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of **A** and a destination of **B** enforces that **A** happens-before **B**. The

inverse relation of happens-after.

## Helper Invocation

A fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations, and which does not have side effects.

## Host

The processor(s) and execution environment that the application runs on, and that the Vulkan API is exposed on.

## Host Mapped Device Memory

Device memory that is mapped for host access using [vkMapMemory](#).

## Host Mapped Foreign Memory

Memory owned by a foreign device that is mapped for host access.

## Host Memory

Memory not accessible to the device, used to store implementation data structures.

## Host-Accessible Subresource

A buffer, or a linear image subresource in either the `VK_IMAGE_LAYOUT_PREINITIALIZED` or `VK_IMAGE_LAYOUT_GENERAL` layout. Host-accessible subresources have a well-defined addressing scheme which can be used by the host.

## Host-Local Memory

Memory that is not local to the device, and **may** be less performant for device access than device-local memory.

## Host-Visible Memory

Device memory that **can** be mapped on the host and **can** be read and written by the host.

## Identically Defined Objects

Objects of the same type where all arguments to their creation or allocation functions, with the exception of `pAllocator`, are

1. Vulkan handles which refer to the same object or
2. identical scalar or enumeration values or
3. Host pointers which point to an array of values or structures which also satisfy these three constraints.

## Image

A resource that represents a multi-dimensional formatted interpretation of device memory. Represented by a [VkImage](#) object.

## Image Subresource

A specific mipmap level and layer of an image.

## **Image Subresource Range**

A set of image subresources that are contiguous mipmap levels and layers.

## **Image View**

An object that represents an image subresource range of a specific image, and state that controls how the contents are interpreted. Represented by a [VkImageView](#) object.

## **Immutable Sampler**

A sampler descriptor provided at descriptor set layout creation time, and that is used for that binding in all descriptor sets allocated from the layout, and cannot be changed.

## **Implicit chroma reconstruction**

An implementation of sampler Y'CbCr conversion which reconstructs the reduced-resolution chroma samples directly at the sample point, as part of the normal texture sampling operation. This is distinct from an *explicit chroma reconstruction* implementation, which reconstructs the reduced-resolution chroma samples to the resolution of the luma samples, then filters the result as part of texture sample interpolation.

## **Implicitly-Enabled Layer**

A layer enabled by a loader-defined mechanism outside the Vulkan API, rather than explicitly by the application during instance or device creation.

## **Index Buffer**

A buffer bound via [vkCmdBindIndexBuffer](#) which is the source of index values used to fetch vertex attributes for a [vkCmdDrawIndexed](#) or [vkCmdDrawIndexedIndirect](#) command.

## **Indexed Drawing Commands**

*Drawing commands* which use an *index buffer* as the source of index values used to fetch vertex attributes for a drawing command. Includes [vkCmdDrawIndexed](#), [vkCmdDrawIndexedIndirectCountKHR](#), [vkCmdDrawIndexedIndirectCountAMD](#), and [vkCmdDrawIndexedIndirect](#).

## **Indirect Commands**

Drawing or dispatching commands that source some of their parameters from structures in buffer memory. Includes [vkCmdDrawIndirect](#), [vkCmdDrawIndexedIndirect](#), [vkCmdDrawIndirectCountKHR](#), [vkCmdDrawIndirectCountAMD](#), [vkCmdDrawMeshTasksIndirectNV](#), [vkCmdDrawMeshTasksIndirectCountNV](#), and [vkCmdDispatchIndirect](#).

## **Indirect Commands Layout**

A definition of a sequence of commands, that are generated on the device via [vkCmdProcessCommandsNVX](#). Each sequence is comprised of multiple [VkIndirectCommandsTokenTypeNVX](#), which represent a subset of traditional command buffer commands. Represented as [VkIndirectCommandsLayoutNVX](#).

## **Indirect Drawing Commands**

*Drawing commands* that source some of their parameters from structures in buffer memory.

Includes [vkCmdDrawIndirect](#), [vkCmdDrawIndirectCountKHR](#),  
[vkCmdDrawIndexedIndirectCountKHR](#), [vkCmdDrawIndirectCountAMD](#),  
[vkCmdDrawIndexedIndirectCountAMD](#), [vkCmdDrawMeshTasksIndirectNV](#),  
[vkCmdDrawMeshTasksIndirectCountNV](#), and [vkCmdDrawIndexedIndirect](#).

## Initial State (Command Buffer)

A command buffer that has not begun recording commands. See also Recorded State and Executable State.

## Inline Uniform Block

A descriptor type that represents uniform data stored directly in descriptor sets, and supports read-only access in a shader.

## Input Attachment

A descriptor type that represents an image view, and supports unfiltered read-only access in a shader, only at the fragment's location in the view.

## Instance

The top-level Vulkan object, which represents the application's connection to the implementation. Represented by a [VkInstance](#) object.

## Instance-Level Command

Any command that is dispatched from an instance, or from a child object of an instance, except for physical devices and their children.

## Instance-Level Functionality

All instance-level commands and objects, and their structures, enumerated types, and enumerants.

## Instance-Level Object

High-level Vulkan objects, which are not physical devices, nor children of physical devices. For example, [VkInstance](#) is an instance-level object.

## Instance (Memory)

In a logical device representing more than one physical device, some device memory allocations have the requested amount of memory allocated multiple times, once for each physical device in a device mask. Each such replicated allocation is an instance of the device memory.

## Instance (Resource)

In a logical device representing more than one physical device, buffer and image resources exist on all physical devices but **can** be bound to memory differently on each. Each such replicated resource is an instance of the resource.

## Internal Synchronization

A type of synchronization **required** of the implementation, where parameters not defined to be externally synchronized **may** require internal mutexing to avoid multithreaded race conditions.

## Invocation (Shader)

A single execution of an entry point in a SPIR-V module. For example, a single vertex's execution of a vertex shader or a single fragment's execution of a fragment shader.

## Invocation Group

A set of shader invocations that are executed in parallel and that **must** execute the same control flow path in order for control flow to be considered dynamically uniform.

## Linear Resource

A resource is *linear* if it is one of the following:

- a [VkBuffer](#)
- a [VkImage](#) created with `VK_IMAGE_TILING_LINEAR`
- a [VkImage](#) created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and whose [Linux DRM format modifier](#) is `DRM_FORMAT_MOD_LINEAR`

A resource is *non-linear* if it is one of the following:

- a [VkImage](#) created with `VK_IMAGE_TILING_OPTIMAL`
- a [VkImage](#) created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and whose [Linux DRM format modifier](#) is not `DRM_FORMAT_MOD_LINEAR`

## Linux DRM Format Modifier

A 64-bit, vendor-prefixed, semi-opaque unsigned integer describing vendor-specific details of an image's memory layout. In Linux graphics APIs, *modifiers* are commonly used to specify the memory layout of externally shared images. An image has a *modifier* if and only if it is created with `tiling` equal to `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`. For more details, refer to the appendix for extension [VK\\_EXT\\_image\\_drm\\_format\\_modifier](#).

## Local Workgroup

A collection of compute shader invocations invoked by a single dispatch command, which share data via [WorkgroupLocal](#) variables and can synchronize with each other.

## Logical Device

An object that represents the application's interface to the physical device. The logical device is the parent of most Vulkan objects. Represented by a [VkDevice](#) object.

## Logical Operation

Bitwise operations between a fragment color value and a value in a color attachment, that produce a final color value to be written to the attachment.

## Lost Device

A state that a logical device **may** be in as a result of unrecoverable implementation errors, or other exceptional conditions.

## Mappable

See Host-Visible Memory.

## Memory Dependency

A memory dependency is an execution dependency which includes availability and visibility operations such that:

- The first set of operations happens-before the availability operation
- The availability operation happens-before the visibility operation
- The visibility operation happens-before the second set of operations

## Memory Domain

A memory domain is an abstract place to which memory writes are made available by availability operations and memory domain operations. The memory domains correspond to the set of agents that the write **can** then be made visible to. The memory domains are *host*, *device*, *shader*, *workgroup instance* (for workgroup instance there is a unique domain for each compute workgroup) and *subgroup instance* (for subgroup instance there is a unique domain for each subgroup).

## Memory Domain Operation

An operation that makes the writes that are available to one memory domain available to another memory domain.

## Memory Heap

A region of memory from which device memory allocations **can** be made.

## Memory Type

An index used to select a set of memory properties (e.g. mappable, cached) for a device memory allocation.

## Mesh Shading Pipeline

A graphics pipeline where the primitives are assembled explicitly in the shader stages. In contrast to the primitive shading pipeline where input primitives are assembled by fixed function processing.

## Mesh Tasks Drawing Commands

*Drawing commands* which create shader invocations organized in workgroups for drawing mesh tasks. Includes [vkCmdDrawMeshTasksNV](#), [vkCmdDrawMeshTasksIndirectNV](#), and [vkCmdDrawMeshTasksIndirectCountNV](#).

## Minimum Miplevel Size

The smallest size that is permitted for a miplevel. For conventional images this is 1x1x1. For corner-sampled images, this is 2x2x2. See [Image Miplevel Sizing](#).

## Mip Tail Region

The set of mipmap levels of a sparse residency texture that are too small to fill a sparse block, and that **must** all be bound to memory collectively and opaquely.

## Multi-planar

A *multi-planar format* (or “planar format”) is an image format consisting of more than one *plane*,

identifiable with a `_2PLANE` or `_3PLANE` component to the format name and listed in [Formats requiring sampler Y<sub>B</sub>C<sub>R</sub> conversion for VK\\_IMAGE\\_ASPECT\\_COLOR\\_BIT image views](#). A *multi-planar image* (or “planar image”) is an image of a multi-planar format.

## Non-Dispatchable Handle

A handle of an integer handle type. Handle values **may** not be unique, even for two objects of the same type.

## Non-Indexed Drawing Commands

*Drawing commands* for which the vertex attributes are sourced in linear order from the vertex input attributes for a drawing command (i.e. they do not use an *index buffer*). Includes `vkCmdDraw`, `vkCmdDrawIndirectCountKHR`, `vkCmdDrawIndirectCountAMD`, and `vkCmdDrawIndirect`.

## Normalized

A value that is interpreted as being in the range [0,1] as a result of being implicitly divided by some other value.

## Normalized Device Coordinates

A coordinate space after perspective division is applied to clip coordinates, and before the viewport transformation converts to framebuffer coordinates.

## Object Table

A binding table for various resources ([VkPipeline](#), [VkBuffer](#), [VkDescriptorSet](#)), so that they can be referenced in device-generated command processing. Represented as [VkObjectTableNVX](#). Entries are registered or unregistered via `uint32_t` indices.

## Obsoleted (feature)

A feature is obsolete if it can no longer be used.

## Opaque Capture Address

A 64-bit value representing the device address of a buffer or memory object that is expected to be used by trace capture/replay tools in combination with the [bufferDeviceAddress](#) feature.

## Overlapped Range (Aliased Range)

The aliased range of a device memory allocation that intersects a given image subresource of an image or range of a buffer.

## Ownership (Resource)

If an entity (e.g. a queue family) has ownership of a resource, access to that resource is well-defined for access by that entity.

## Packed Format

A format whose components are stored as a single texel block in memory, with their relative locations defined within that element.

## Passthrough Geometry Shader

A geometry shader which uses the `PassthroughNV` decoration on a variable in its input interface.

Output primitives in a passthrough geometry shader always have the same topology as the input primitive and are not produced by emitting vertices.

## Payload

Importable or exportable reference to the internal data of an object in Vulkan.

## Per-View

A variable that has an array of values which are output, one for each view that is being generated. A mesh shader which uses the `PerViewNV` decoration on a variable in its output interface.

## Peer Memory

An instance of memory corresponding to a different physical device than the physical device performing the memory access, in a logical device that represents multiple physical devices.

## Physical Device

An object that represents a single device in the system. Represented by a `VkPhysicalDevice` object.

## Physical-Device-Level Command

Any command that is dispatched from a physical device.

## Physical-Device-Level Functionality

All physical-device-level commands and objects, and their structures, enumerated types, and enumerants.

## Physical-Device-Level Object

Physical device objects. For example, `VkPhysicalDevice` is a physical-device-level object.

## Pipeline

An object that controls how graphics or compute work is executed on the device. A pipeline includes one or more shaders, as well as state controlling any non-programmable stages of the pipeline. Represented by a `VkPipeline` object.

## Pipeline Barrier

An execution and/or memory dependency recorded as an explicit command in a command buffer, that forms a dependency between the previous and subsequent commands.

## Pipeline Cache

An object that **can** be used to collect and retrieve information from pipelines as they are created, and **can** be populated with previously retrieved information in order to accelerate pipeline creation. Represented by a `VkPipelineCache` object.

## Pipeline Layout

An object that defines the set of resources (via a collection of descriptor set layouts) and push constants used by pipelines that are created using the layout. Used when creating a pipeline and when binding descriptor sets and setting push constant values. Represented by a `VkPipelineLayout` object.

## Pipeline Stage

A logically independent execution unit that performs some of the operations defined by an action command.

## pNext Chain

A set of structures [chained together](#) through their pNext members.

## Planar

See *multi-planar*.

## Plane

An *image plane* is part of the representation of an image, containing a subset of the color channels required to represent the texels in the image and with a contiguous mapping of coordinates to bound memory. Most images consist only of a single plane, but some formats spread the channels across multiple image planes. The host-accessible properties of each image plane are accessed in a linear layout using [vkGetImageSubresourceLayout](#). If a multi-planar image is created with the [VK\\_IMAGE\\_CREATE\\_DISJOINT\\_BIT](#) bit set, the image is described as *disjoint*, and its planes are therefore bound to memory independently.

## Point Sampling (Rasterization)

A rule that determines whether a fragment sample location is covered by a polygon primitive by testing whether the sample location is in the interior of the polygon in framebuffer-space, or on the boundary of the polygon according to the tie-breaking rules.

## Presentable image

A [VkImage](#) object obtained from a [VkSwapchainKHR](#) used to present to a [VkSurfaceKHR](#) object.

## Preserve Attachment

One of a list of attachments in a subpass description that is not read or written by the subpass, but that is read or written on earlier and later subpasses and whose contents **must** be preserved through this subpass.

## Primary Command Buffer

A command buffer that **can** execute secondary command buffers, and **can** be submitted directly to a queue.

## Primitive Shading Pipeline

A graphics pipeline where input primitives are assembled by fixed function processing. It is the counterpart to mesh shading.

## Primitive Topology

State that controls how vertices are assembled into primitives, e.g. as lists of triangles, strips of lines, etc..

## Promoted (feature)

A feature from an older extension is considered promoted if it is made available as part of a new core version or newer extension with wider support.

## **Protected Buffer**

A buffer to which protected device memory **can** be bound.

## **Protected-capable Device Queue**

A device queue to which protected command buffers **can** be submitted.

## **Protected Command Buffer**

A command buffer which **can** be submitted to a protected-capable device queue.

## **Protected Device Memory**

Device memory which **can** be visible to the device but **must** not be visible to the host.

## **Protected Image**

An image to which protected device memory **can** be bound.

## **Provisional**

A feature is released provisionally in order to get wider feedback on the functionality before it is finalized. Provisional features may change in ways that break backwards compatibility, and thus are not recommended for use in production applications.

## **Provoking Vertex**

The vertex in a primitive from which flat shaded attribute values are taken. This is generally the “first” vertex in the primitive, and depends on the primitive topology.

## **Push Constants**

A small bank of values writable via the API and accessible in shaders. Push constants allow the application to set values used in shaders without creating buffers or modifying and binding descriptor sets for each update.

## **Push Constant Interface**

The set of variables with **PushConstant** storage class that are statically used by a shader entry point, and which receive values from push constant commands.

## **Push Descriptors**

Descriptors that are written directly into a command buffer rather than into a descriptor set. Push descriptors allow the application to set descriptors used in shaders without allocating or modifying descriptor sets for each update.

## **Descriptor Update Template**

An object that specifies a mapping from descriptor update information in host memory to elements in a descriptor set, which helps enable more efficient descriptor set updates.

## **Query Pool**

An object containing a number of query entries and their associated state and results. Represented by a [VkQueryPool](#) object.

## **Queue**

An object that executes command buffers and sparse binding operations on a device.

Represented by a [VkQueue](#) object.

## Queue Family

A set of queues that have common properties and support the same functionality, as advertised in [VkQueueFamilyProperties](#).

## Queue Operation

A unit of work to be executed by a specific queue on a device, submitted via a [queue submission command](#). Each queue submission command details the specific queue operations that occur as a result of calling that command. Queue operations typically include work that is specific to each command, and synchronization tasks.

## Queue Submission

Zero or more batches and an optional fence to be signaled, passed to a command for execution on a queue. See the [Devices and Queues chapter](#) for more information.

## Recording State (Command Buffer)

A command buffer that is ready to record commands. See also Initial State and Executable State.

## Release Operation (Resource)

An operation that releases ownership of an image subresource or buffer range.

## Render Pass

An object that represents a set of framebuffer attachments and phases of rendering using those attachments. Represented by a [VkRenderPass](#) object.

## Render Pass Instance

A use of a render pass in a command buffer.

## Required Extensions

Extensions that **must** be enabled alongside extensions dependent on them (see [Extension Dependencies](#)).

## Reset (Command Buffer)

Resetting a command buffer discards any previously recorded commands and puts a command buffer in the initial state.

## Residency Code

An integer value returned by sparse image instructions, indicating whether any sparse unbound texels were accessed.

## Resolve Attachment

A subpass attachment point, or image view, that is the target of a multisample resolve operation from the corresponding color attachment at the end of the subpass.

## Retired Swapchain

A swapchain that has been used as the `oldSwapchain` parameter to [vkCreateSwapchainKHR](#). Images cannot be acquired from a retired swapchain, however images that were acquired (but

not presented) before the swapchain was retired **can** be presented.

## Sample Shading

Invoking the fragment shader multiple times per fragment, with the covered samples partitioned among the invocations.

## Sampled Image

A descriptor type that represents an image view, and supports filtered (sampled) and unfiltered read-only access in a shader.

## Sampler

An object containing state that controls how sampled image data is sampled (or filtered) when accessed in a shader. Also a descriptor type describing the object. Represented by a [VkSampler](#) object.

## Secondary Command Buffer

A command buffer that **can** be executed by a primary command buffer, and **must** not be submitted directly to a queue.

## Self-Dependency

A subpass dependency from a subpass to itself, i.e. with `srcSubpass` equal to `dstSubpass`. A self-dependency is not automatically performed during a render pass instance, rather a subset of it **can** be performed via [vkCmdPipelineBarrier](#) during the subpass.

## Semaphore

A synchronization primitive that supports signal and wait operations, and **can** be used to synchronize operations within a queue or across queues. Represented by a [VkSemaphore](#) object.

## Shader

Instructions selected (via an entry point) from a shader module, which are executed in a shader stage.

## Shader Code

A stream of instructions used to describe the operation of a shader.

## Shader Module

A collection of shader code, potentially including several functions and entry points, that is used to create shaders in pipelines. Represented by a [VkShaderModule](#) object.

## Shader Stage

A stage of the graphics or compute pipeline that executes shader code.

## Shading Rate

The ratio of the number of fragment shader invocations generated in a fully covered framebuffer region to the size (in pixels) of that region.

## Shading Rate Image

An image used to establish the shading rate for a framebuffer region, where each pixel controls

the shading rate for a corresponding framebuffer region.

### Shared presentable image

A presentable image created from a swapchain with `VkPresentModeKHR` set to either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`.

### Side Effect

A store to memory or atomic operation on memory from a shader invocation.

### Single-plane format

A format that is not *multi-planar*.

### Size-Compatible Image Formats

When a compressed image format and an uncompressed image format are size-compatible, it means that the texel block size of the uncompressed format **must** equal the texel block size of the compressed format.

### Sparse Block

An element of a sparse resource that can be independently bound to memory. Sparse blocks of a particular sparse resource have a corresponding size in bytes that they use in the bound memory.

### Sparse Image Block

A sparse block in a sparse partially-resident image. In addition to the sparse block size in bytes, sparse image blocks have a corresponding width, height, and depth that define the dimensions of these elements in units of texels or compressed texel blocks, the latter being used in case of sparse images having a block-compressed format.

### Sparse Unbound Texel

A texel read from a region of a sparse texture that does not have memory bound to it.

### Static Use

An object in a shader is statically used by a shader entry point if any function in the entry point's call tree contains an instruction using the object. Static use is used to constrain the set of descriptors used by a shader entry point.

### Storage Buffer

A descriptor type that represents a buffer, and supports reads, writes, and atomics in a shader.

### Storage Image

A descriptor type that represents an image view, and supports unfiltered loads, stores, and atomics in a shader.

### Storage Texel Buffer

A descriptor type that represents a buffer view, and supports unfiltered, formatted reads, writes, and atomics in a shader.

### Subgroup

A set of shader invocations that **can** synchronize and share data with each other efficiently. In compute shaders, the *local workgroup* is a superset of the subgroup.

## Subgroup Mask

A bitmask for all invocations in the current subgroup with one bit per invocation, starting with the least significant bit in the first vector component, continuing to the last bit (less than [SubgroupSize](#)) in the last required vector component.

## Subpass

A phase of rendering within a render pass, that reads and writes a subset of the attachments.

## Subpass Dependency

An execution and/or memory dependency between two subpasses described as part of render pass creation, and automatically performed between subpasses in a render pass instance. A subpass dependency limits the overlap of execution of the pair of subpasses, and **can** provide guarantees of memory coherence between accesses in the subpasses.

## Subpass Description

Lists of attachment indices for input attachments, color attachments, depth/stencil attachment, resolve attachments, depth/stencil resolve, and preserve attachments used by the subpass in a render pass.

## Subset (Self-Dependency)

A subset of a self-dependency is a pipeline barrier performed during the subpass of the self-dependency, and whose stage masks and access masks each contain a subset of the bits set in the identically named mask in the self-dependency.

## Texel Block

A single addressable element of an image with an uncompressed [VkFormat](#), or a single compressed block of an image with a compressed [VkFormat](#).

## Texel Block Size

The size (in bytes) used to store a texel block of a compressed or uncompressed image.

## Texel Coordinate System

One of three coordinate systems (normalized, unnormalized, integer) that define how texel coordinates are interpreted in an image or a specific mipmap level of an image.

## Timeline Semaphore

A semaphore with a monotonically increasing 64-bit unsigned integer payload indicating whether the semaphore is signaled with respect to a particular reference value. Represented by a [VkSemaphore](#) object created with a semaphore type of [VK\\_SEMAPHORE\\_TYPE\\_TIMELINE\\_KHR](#).

## Uniform Texel Buffer

A descriptor type that represents a buffer view, and supports unfiltered, formatted, read-only access in a shader.

## Uniform Buffer

A descriptor type that represents a buffer, and supports read-only access in a shader.

## Units in the Last Place (ULP)

A measure of floating-point error loosely defined as the smallest representable step in a floating-point format near a given value. For the precise definition see [Precision and Operation of SPIR-V instructions](#) or Jean-Michel Muller, “On the definition of  $\text{ulp}(x)$ ”, RR-5504, INRIA. Other sources may also use the term “unit of least precision”.

## Unnormalized

A value that is interpreted according to its conventional interpretation, and is not normalized.

## Unprotected Buffer

A buffer to which unprotected device memory **can** be bound.

## Unprotected Command Buffer

A command buffer which **can** be submitted to an unprotected device queue or a protected-capable device queue.

## Unprotected Device Memory

Device memory which **can** be visible to the device and **can** be visible to the host.

## Unprotected Image

An image to which unprotected device memory **can** be bound.

## User-Defined Variable Interface

A shader entry point’s variables with [Input](#) or [Output](#) storage class that are not built-in variables.

## Vertex Input Attribute

A graphics pipeline resource that produces input values for the vertex shader by reading data from a vertex input binding and converting it to the attribute’s format.

## Vertex Stream

A vertex stream is where the last vertex processing stage outputs vertex data, which then goes to the rasterizer, is captured to a transform feedback buffer, or both. Geometry shaders **can** emit primitives to multiple independent vertex streams. Each vertex emitted by the geometry shader is directed at one of the vertex streams.

## Validation Cache

An object that **can** be used to collect and retrieve validation results from the validation layers, and **can** be populated with previously retrieved results in order to accelerate the validation process. Represented by a [VkValidationCacheEXT](#) object.

## Vertex Input Binding

A graphics pipeline resource that is bound to a buffer and includes state that affects addressing calculations within that buffer.

## Vertex Input Interface

A vertex shader entry point’s variables with [Input](#) storage class, which receive values from

vertex input attributes.

## **Vertex Processing Stages**

A set of shader stages that comprises the vertex shader, tessellation control shader, tessellation evaluation shader, and geometry shader stages. The task and mesh shader stages also belong to this group.

## **View Mask**

When multiview is enabled, a view mask is a property of a subpass controlling which views the rendering commands are broadcast to.

## **View Volume**

A subspace in homogeneous coordinates, corresponding to post-projection x and y values between -1 and +1, and z values between 0 and +1.

## **Viewport Transformation**

A transformation from normalized device coordinates to framebuffer coordinates, based on a viewport rectangle and depth range.

## **Visibility Operation**

An operation that causes available values to become visible to specified memory accesses.

## **Visible**

A state of values written to memory that allows them to be accessed by a set of operations.

# Common Abbreviations

Abbreviations and acronyms are sometimes used in the Specification and the API where they are considered clear and commonplace, and are defined here:

## **Src**

Source

## **Dst**

Destination

## **Min**

Minimum

## **Max**

Maximum

## **Rect**

Rectangle

## **Info**

Information

## **LOD**

Level of Detail

## **ID**

Identifier

## **UUID**

Universally Unique Identifier

## **Op**

Operation

## **R**

Red color component

## **G**

Green color component

## **B**

Blue color component

## **A**

Alpha color component

**RTZ**

Round towards zero

**RTE**

Round to nearest even

# Prefixes

Prefixes are used in the API to denote specific semantic meaning of Vulkan names, or as a label to avoid name clashes, and are explained here:

## VK/Vk/vk

Vulkan namespace

All types, commands, enumerants and defines in this specification are prefixed with these two characters.

## PFN/pfn

Function Pointer

Denotes that a type is a function pointer, or that a variable is of a pointer type.

## p

Pointer

Variable is a pointer.

## vkCmd

Commands that record commands in command buffers

These API commands do not result in immediate processing on the device. Instead, they record the requested action in a command buffer for execution when the command buffer is submitted to a queue.

## s

Structure

Used to denote the `VK_STRUCTURE_TYPE*` member of each structure in `sType`

# Appendix H: Credits (Informative)

Vulkan 1.1 is the result of contributions from many people and companies participating in the Khronos Vulkan Working Group, as well as input from the Vulkan Advisory Panel.

Members of the Working Group, including the company that they represented at the time of their most recent contribution, are listed in the following sections. Some specific contributions made by individuals are listed together with their name.

## Working Group Contributors to Vulkan 1.1 and 1.0

- Adam Jackson, Red Hat
- Alexander Galazin, Arm
- Alex Bourd, Qualcomm Technologies, Inc.
- Alon Or-bach, Samsung Electronics (WSI technical sub-group chair)
- Andrew Garrard, Samsung Electronics (format wrangler)
- Andrew Woloszyn, Google
- Antoine Labour, Google
- Bill Licea-Kane, Qualcomm Technologies, Inc.
- Cass Everitt, Oculus VR
- Chad Versace, Google
- Christophe Riccio, Unity Technologies
- Dan Baker, Oxide Games
- Dan Ginsburg, Valve Software
- Daniel Johnston, Intel
- Daniel Koch, NVIDIA ([Shader Interfaces; Features, Limits, and Formats](#))
- Daniel Rakos, AMD
- David Airlie, Red Hat
- David Miller, Miller & Mattson (Vulkan reference card)
- David Neto, Google
- Dominik Witczak, AMD
- Graeme Leese, Broadcom
- Graham Sellers, AMD
- Ian Romanick, Intel
- James Jones, NVIDIA
- Jan-Harald Fredriksen, Arm
- Jan Hermes, Continental Corporation

- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA (extensive contributions, exhaustive review and rewrites for technical correctness)
- Jeff Juliano, NVIDIA
- Jesse Barker, Unity Technologies
- Jesse Hall, Google
- Johannes van Waveren, Oculus VR
- John Kessenich, Google (SPIR-V and GLSL for Vulkan spec author)
- John McDonald, Valve Software
- Jonas Gustavsson, Samsung Electronics
- Jon Ashburn, LunarG
- Jon Leech, Independent (XML toolchain, normative language, release wrangler)
- Jungwoo Kim, Samsung Electronics
- Kathleen Mattson, Miller & Mattson (Vulkan reference card)
- Kenneth Benzie, Codeplay Software Ltd.
- Kerch Holt, NVIDIA (SPIR-V technical sub-group chair)
- Kristian Kristensen, Intel
- Mark Lobodzinski, LunarG
- Mathias Heyer, NVIDIA
- Mathias Schott, NVIDIA
- Maurice Ribble, Qualcomm Technologies, Inc.
- Michael Worcester, Imagination Technologies
- Mika Isojarvi, Google
- Mitch Singer, AMD
- Neil Henning, Codeplay Software Ltd.
- Neil Trevett, NVIDIA
- Norbert Nopper, Independent
- Pierre Boudier, NVIDIA
- Pierre-Loup Griffais, Valve Software
- Piers Daniell, NVIDIA (dynamic state, copy commands, memory types)
- Pyry Haulos, Google (Vulkan conformance test subcommittee chair)
- Ray Smith, Arm
- Robert Simpson, Qualcomm Technologies, Inc.
- Rolando Caloca Olivares, Epic Games
- Sean Harmer, KDAB Group

- Shannon Woods, Google
- Slawomir Cygan, Intel
- Slawomir Grajewski, Intel
- Stuart Smith, Imagination Technologies
- Timothy Lottes, AMD
- Tobias Hector, Imagination Technologies (validity language and toolchain)
- Tom Olson, Arm (working group chair)
- Tony Barbour, LunarG
- Yanjun Zhang, VeriSilicon

## Working Group Contributors to Vulkan 1.1

- Aaron Greig, Codeplay Software Ltd.
- Aaron Hagan, AMD
- Alan Ward, Google
- Alejandro Piñeiro, Igalia
- Andres Gomez, Igalia
- Baldur Karlsson, Independent
- Barthold Lichtenbelt, NVIDIA
- Bas Nieuwenhuizen, Google
- Bill Hollings, Brenwill
- Colin Riley, AMD
- Cort Stratton, Google
- Courtney Goeltzenleuchter, Google
- Dae Kim, Imagination Technologies
- Daniel Stone, Collabora
- David Pinedo, LunarG
- Dejan Mircevski, Google
- Dzmitry Malyshau, Mozilla
- Erika Johnson, LunarG
- Greg Fischer, LunarG
- Hans-Kristian Arntzen, Arm
- Iago Toral, Igalia
- Ian Elliott, Google
- Jeff Leger, Qualcomm Technologies, Inc.

- Jeff Vigil, Samsung Electronics
- Jens Owen, Google
- Joe Davis, Samsung Electronics
- John Zulauf, LunarG
- Jordan Justen, Intel
- Jörg Wagner, Arm
- Kalle Raita, Google
- Karen Ghavam, LunarG
- Karl Schultz, LunarG
- Kenneth Russell, Google
- Kevin O’Neil, AMD
- Lauri Ilola, Nokia
- Lenny Komow, LunarG
- Lionel Landwerlin, Intel
- Maciej Jesionowski, AMD
- Mais Alnasser, AMD
- Marcin Rogucki, Mobica
- Mark Callow, Independent
- Mark Kilgard, NVIDIA
- Markus Tavenrath, NVIDIA
- Mark Young, LunarG
- Matthäus Chajdas, AMD
- Matt Netsch, Qualcomm Technologies, Inc.
- Michael O’Hara, AMD
- Michael Wong, Codeplay Software Ltd.
- Mike Schuchardt, LunarG
- Mike Weiblen, LunarG
- Nicolai Hähnle, AMD
- Nuno Subtil, NVIDIA
- Patrick Cozzi, Independent
- Petros Bantolas, Imagination Technologies
- Ralph Potter, Codeplay Software Ltd.
- Rob Barris, NVIDIA
- Ruihao Zhang, Qualcomm Technologies, Inc.
- Sorel Bosan, AMD

- Stephen Huang, Mediatek
- Tilmann Scheller, Samsung Electronics
- Tomasz Bednarz, Independent
- Victor Eruhimov, ???
- Wolfgang Engel, ???

## Working Group Contributors to Vulkan 1.0

- Adam Śmigielski, Mobica
- Allen Hux, Intel
- Andrew Cox, Samsung Electronics
- Andrew Poole, Samsung Electronics
- Andrew Rafter, Samsung Electronics
- Andrew Richards, Codeplay Software Ltd.
- Aras Pranckevičius, Unity Technologies
- Ashwin Kolhe, NVIDIA
- Ben Bowman, Imagination Technologies
- Benj Lipchak
- Bill Hollings, The Brenwill Workshop
- Brent E. Insko, Intel
- Brian Ellis, Qualcomm Technologies, Inc.
- Cemil Azizoglu, Canonical
- Chang-Hyo Yu, Samsung Electronics
- Chia-I Wu, LunarG
- Chris Frascati, Qualcomm Technologies, Inc.
- Cody Northrop, LunarG
- Courtney Goeltzenleuchter, LunarG
- Damien Leone, NVIDIA
- David Mao, AMD
- David Yu, Pixar
- Frank (LingJun) Chen, Qualcomm Technologies, Inc.
- Fred Liao, Mediatek
- Gabe Dagani, Freescale
- Graham Connor, Imagination Technologies
- Hwanyong Lee, Kyungpook National University

- Ian Elliott, LunarG
- James Hughes, Oculus VR
- Jeff Vigil, Qualcomm Technologies, Inc.
- Jens Owen, LunarG
- Jeremy Hayes, LunarG
- Jonathan Hamilton, Imagination Technologies
- Krzysztof Iwanicki, Samsung Electronics
- Larry Seiler, Intel
- Lutz Latta, Lucasfilm
- Maria Rovatsou, Codeplay Software Ltd.
- Mark Callow
- Mateusz Przybylski, Intel
- Maxim Lukyanov, Samsung Electronics
- Michael Lentine, Google
- Michal Pietrasik, Intel
- Mike Stroyan, LunarG
- Minyoung Son, Samsung Electronics
- Mythri Venugopal, Samsung Electronics
- Naveen Leekha, Google
- Nick Penwarden, Epic Games
- Niklas Smedberg, Unity Technologies
- Pat Brown, NVIDIA
- Patrick Doane, Blizzard Entertainment
- Peter Lohrmann, Valve Software
- Piotr Bialecki, Intel
- Prabindh Sundareson, Samsung Electronics
- Rob Stepinski, Transgaming
- Roy Ju, Mediatek
- Rufus Hamade, Imagination Technologies
- Sean Ellis, Arm
- Stefanus Du Toit, Google
- Steve Hill, Broadcom
- Steve Viggers, Core Avionics & Industrial Inc.
- Tim Foley, Intel
- Timo Suoranta, AMD

- Tobin Ehlis, LunarG
- Tomasz Kubale, Intel
- Wayne Lister, Imagination Technologies

## Other Credits

The Vulkan Advisory Panel members provided important real-world usage information and advice that helped guide design decisions.

The wider Vulkan community have provided useful feedback, questions and spec changes that have helped improve the quality of the Specification via [GitHub](#).

Administrative support to the Working Group for Vulkan 1.1 was provided by Khronos staff including Angela Cheng, Ann Thorsnes, Emily Stearns, Liz Maitral, and Dominic Agoro-Ombaka; and by Alex Crabb of Caster Communications.

Administrative support for Vulkan 1.0 was provided by Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, Kathleen Mattson and Michelle Clark of Gold Standard Group.

Technical support was provided by James Riordon, webmaster of Khronos.org and OpenGL.org.