# Vulkan® 1.0.49 - A Specification (with all registered Vulkan extensions)

The Khronos Vulkan Working Group

Version 1.0.49, Sun, 21 May 2017 00:01:50 +0000

# Table of Contents


1. Introduction . . . 3
    1.1. What is the Vulkan Graphics System? . . . 3
    1.2. Filing Bug Reports . . . 4
    1.3. Terminology . . . 4
    1.4. Normative References . . . 5
2. Fundamentals . . . 6
    2.1. Architecture Model . . . 6
    2.2. Execution Model . . . 6
    2.3. Object Model . . . 9
    2.4. Command Syntax and Duration . . . 12
    2.5. Threading Behavior . . . 13
    2.6. Errors . . . 22
    2.7. Numeric Representation and Computation . . . 28
    2.8. Fixed-Point Data Conversions . . . 30
    2.9. API Version Numbers and Semantics . . . 31
    2.10. Common Object Types . . . 32
3. Initialization . . . 34
    3.1. Command Function Pointers . . . 34
    3.2. Instances . . . 36
4. Devices and Queues . . . 42
    4.1. Physical Devices . . . 42
    4.2. Devices . . . 50
    4.3. Queues . . . 59
5. Command Buffers . . . 63
    5.1. Command Buffer Lifecycle . . . 63
    5.2. Command Pools . . . 64
    5.3. Command Buffer Allocation and Management . . . 70
    5.4. Command Buffer Recording . . . 74
    5.5. Command Buffer Submission . . . 79
    5.6. Queue Forward Progress . . . 90
    5.7. Secondary Command Buffer Execution . . . 91
    5.8. Command Buffer Device Mask . . . 94
6. Synchronization and Cache Control . . . 97
    6.1. Execution and Memory Dependencies . . . 97
    6.2. Implicit Synchronization Guarantees . . . 110
    6.3. Fences . . . 111
    6.4. Semaphores . . . 121
    6.5. Events . . . 135

6.6. Pipeline Barriers . . . 147
6.7. Memory Barriers . . . 152
6.8. Wait Idle Operations . . . 160
6.9. Host Write Ordering Guarantees . . . 162
6.10. Synchronization and Multiple Physical Devices . . . 162
7. Render Pass . . . 163
7.1. Render Pass Creation . . . 164
7.2. Render Pass Compatibility . . . 185
7.3. Framebuffers . . . 185
7.4. Render Pass Commands . . . 190
8. Shaders . . . 199
8.1. Shader Modules . . . 199
8.2. Shader Execution . . . 202
8.3. Shader Memory Access Ordering . . . 202
8.4. Shader Inputs and Outputs . . . 204
8.5. Vertex Shaders . . . 204
8.6. Tessellation Control Shaders . . . 205
8.7. Tessellation Evaluation Shaders . . . 205
8.8. Geometry Shaders . . . 206
8.9. Fragment Shaders . . . 206
8.10. Compute Shaders . . . 207
8.11. Interpolation Decorations . . . 207
8.12. Static Use . . . 208
8.13. Invocation and Derivative Groups . . . 208
9. Pipelines . . . 210
9.1. Compute Pipelines . . . 212
9.2. Graphics Pipelines . . . 218
9.3. Pipeline destruction . . . 230
9.4. Multiple Pipeline Creation . . . 231
9.5. Pipeline Derivatives . . . 231
9.6. Pipeline Cache . . . 231
9.7. Specialization Constants . . . 238
9.8. Pipeline Binding . . . 242
10. Memory Allocation . . . 244
10.1. Host Memory . . . 244
10.2. Device Memory . . . 251
11. Resource Creation . . . 285
11.1. Buffers . . . 285
11.2. Buffer Views . . . 292
11.3. Images . . . 295
11.4. Image Layouts . . . 311

11.5. Image Views . . . 314
11.6. Resource Memory Association . . . 325
11.7. Resource Sharing Mode . . . 339
11.8. Memory Aliasing . . . 340
12. Samplers . . . 343
13. Resource Descriptors . . . 350
13.1. Descriptor Types . . . 351
13.2. Descriptor Sets . . . 359
14. Shader Interfaces . . . 410
14.1. Shader Input and Output Interfaces . . . 410
14.2. Vertex Input Interface . . . 414
14.3. Fragment Output Interface . . . 414
14.4. Fragment Input Attachment Interface . . . 415
14.5. Shader Resource Interface . . . 416
14.6. Built-In Variables . . . 421
15. Image Operations . . . 438
15.1. Image Operations Overview . . . 438
15.2. Conversion Formulas . . . 441
15.3. Texel Input Operations . . . 443
15.4. Texel Output Operations . . . 449
15.5. Derivative Operations . . . 450
15.6. Normalized Texel Coordinate Operations . . . 452
15.7. Unnormalized Texel Coordinate Operations . . . 457
15.8. Image Sample Operations . . . 458
15.9. Image Operation Steps . . . 461
16. Queries . . . 463
16.1. Query Pools . . . 463
16.2. Query Operation . . . 466
16.3. Occlusion Queries . . . 477
16.4. Pipeline Statistics Queries . . . 478
16.5. Timestamp Queries . . . 480
17. Clear Commands . . . 484
17.1. Clearing Images Outside A Render Pass Instance . . . 484
17.2. Clearing Images Inside A Render Pass Instance . . . 488
17.3. Clear Values . . . 491
17.4. Filling Buffers . . . 492
17.5. Updating Buffers . . . 494
18. Copy Commands . . . 497
18.1. Common Operation . . . 497
18.2. Copying Data Between Buffers . . . 498
18.3. Copying Data Between Images . . . 500

18.4. Copying Data Between Buffers and Images . . . 507
18.5. Image Copies with Scaling . . . 515
18.6. Resolving Multisample Images . . . 523
19. Drawing Commands . . . 527
19.1. Primitive Topologies . . . 528
19.2. Primitive Order . . . 534
19.3. Programmable Primitive Shading . . . 535
20. Fixed-Function Vertex Processing . . . 560
20.1. Vertex Attributes . . . 560
20.2. Vertex Input Description . . . 565
20.3. Example . . . 570
21. Tessellation . . . 572
21.1. Tessellator . . . 572
21.2. Tessellator Patch Discard . . . 574
21.3. Tessellator Spacing . . . 574
21.4. Tessellation Primitive Ordering . . . 575
21.5. Triangle Tessellation . . . 575
21.6. Quad Tessellation . . . 578
21.7. Isoline Tessellation . . . 581
21.8. Tessellation Pipeline State . . . 581
22. Geometry Shading . . . 583
22.1. Geometry Shader Input Primitives . . . 583
22.2. Geometry Shader Output Primitives . . . 584
22.3. Multiple Invocations of Geometry Shaders . . . 584
22.4. Geometry Shader Primitive Ordering . . . 585
22.5. Geometry Shader Passthrough . . . 585
23. Fixed-Function Vertex Post-Processing . . . 587
23.1. Viewport Swizzle . . . 587
23.2. Flat Shading . . . 589
23.3. Primitive Clipping . . . 590
23.4. Clipping Shader Outputs . . . 592
23.5. Controlling Viewport W Scaling . . . 592
23.6. Coordinate Transformations . . . 595
23.7. Controlling the Viewport . . . 596
24. Rasterization . . . 601
24.1. Discarding Primitives Before Rasterization . . . 604
24.2. Rasterization Order . . . 604
24.3. Multisampling . . . 606
24.4. Sample Shading . . . 607
24.5. Points . . . 607
24.6. Line Segments . . . 608

24.7. Polygons . . . 611
25. Fragment Operations . . . 617
25.1. Early Per-Fragment Tests . . . 617
25.2. Discard Rectangles Test . . . 617
25.3. Scissor Test . . . 620
25.4. Sample Mask . . . 622
25.5. Early Fragment Test Mode . . . 622
25.6. Late Per-Fragment Tests . . . 623
25.7. Multisample Coverage . . . 623
25.8. Depth and Stencil Operations . . . 624
25.9. Depth Bounds Test . . . 625
25.10. Stencil Test . . . 627
25.11. Depth Test . . . 633
25.12. Sample Counting . . . 634
26. The Framebuffer . . . 635
26.1. Blending . . . 635
26.2. Logical Operations . . . 643
26.3. Color Write Mask . . . 644
27. Dispatching Commands . . . 646
28. Device-Generated Commands . . . 654
28.1. Features and Limitations . . . 654
28.2. Binding Object Table . . . 656
28.3. Indirect Commands Layout . . . 666
28.4. Indirect Commands Generation . . . 675
29. Sparse Resources . . . 682
29.1. Sparse Resource Features . . . 682
29.2. Sparse Buffers and Fully-Resident Images . . . 683
29.3. Sparse Partially-Resident Buffers . . . 684
29.4. Sparse Partially-Resident Images . . . 684
29.5. Sparse Memory Aliasing . . . 690
29.6. Sparse Resource Implementation Guidelines . . . 691
29.7. Sparse Resource API . . . 693
29.8. Examples . . . 713
30. Window System Integration (WSI) . . . 719
30.1. WSI Platform . . . 719
30.2. WSI Surface . . . 719
30.3. Presenting Directly to Display Devices . . . 737
30.4. Querying for WSI Support . . . 753
30.5. Surface Queries . . . 758
30.6. Device Group Queries . . . 774
30.7. Display Timing Queries . . . 778

30.8. WSI Swapchain . . . 783
30.9. Hdr Metadata . . . 816
31. Extended Functionality . . . 818
31.1. Layers . . . 818
31.2. Extensions . . . 821
32. Features, Limits, and Formats . . . 826
32.1. Features . . . 826
32.2. Limits . . . 838
32.3. Formats . . . 860
32.4. Additional Image Capabilities . . . 906
32.5. Additional Buffer Capabilities . . . 918
32.6. Optional Semaphore Capabilities . . . 920
33. Debugging . . . 924
33.1. Debug Markers . . . 925
33.2. Debug Report Callbacks . . . 931
Appendix A: Vulkan Environment for SPIR-V . . . 940
Required Versions and Formats . . . 940
Capabilities . . . 940
Validation Rules within a Module . . . 942
Precision and Operation of SPIR-V Instructions . . . 944
Appendix B: Compressed Image Formats . . . 947
Block-Compressed Image Formats . . . 948
ETC Compressed Image Formats . . . 949
ASTC Compressed Image Formats . . . 950
Appendix C: Layers & Extensions . . . 951
VK_KHR_android_surface . . . 951
VK_KHR_descriptor_update_template . . . 954
VK_KHR_display . . . 955
VK_KHR_display_swapchain . . . 962
VK_KHR_get_physical_device_properties2 . . . 966
VK_KHR_get_surface_capabilities2 . . . 969
VK_KHR_incremental_present . . . 971
VK_KHR_maintenance1 . . . 973
VK_KHR_mir_surface . . . 975
VK_KHR_push_descriptor . . . 977
VK_KHR_sampler_mirror_clamp_to_edge . . . 979
VK_KHR_shader_draw_parameters . . . 980
VK_KHR_shared_presentable_image . . . 983
VK_KHR_surface . . . 986
VK_KHR_swapchain . . . 991
VK_KHR_wayland_surface . . . 1005

VK_KHR_win32_surface . . . 1007
VK_KHR_xcb_surface . . . 1010
VK_KHR_xlib_surface . . . 1012
VK_KHX_device_group . . . 1014
VK_KHX_device_group_creation . . . 1017
VK_KHX_external_memory_capabilities . . . 1019
VK_KHX_external_memory . . . 1022
VK_KHX_external_memory_win32 . . . 1026
VK_KHX_external_memory_fd . . . 1028
VK_KHX_win32_keyed_mutex . . . 1030
VK_KHX_external_semaphore_capabilities . . . 1031
VK_KHX_external_semaphore . . . 1033
VK_KHX_external_semaphore_win32 . . . 1034
VK_KHX_external_semaphore_fd . . . 1036
VK_KHX_multiview . . . 1038
VK_EXT_debug_marker . . . 1040
VK_EXT_debug_report . . . 1044
VK_EXT_validation_flags . . . 1049
VK_EXT_direct_mode_display . . . 1050
VK_EXT_acquire_xlib_display . . . 1052
VK_EXT_discard_rectangles . . . 1054
VK_EXT_display_surface_counter . . . 1055
VK_EXT_display_control . . . 1057
VK_EXT_shader_subgroup_ballot . . . 1059
VK_EXT_shader_subgroup_vote . . . 1062
VK_EXT_hdr_metadata . . . 1065
VK_EXT_swapchain_colorspace . . . 1066
VK_AMD_draw_indirect_count . . . 1068
VK_AMD_gcn_shader . . . 1069
VK_AMD_gpu_shader_half_float . . . 1070
VK_AMD_negative_viewport_height . . . 1071
VK_AMD_rasterization_order . . . 1072
VK_AMD_shader_ballot . . . 1074
VK_AMD_shader_explicit_vertex_parameter . . . 1075
VK_AMD_shader_trinary_minmax . . . 1076
VK_AMD_texture_gather_bias_lod . . . 1077
VK_GOOGLE_display_timing . . . 1080
VK_IMG_filter_cubic . . . 1082
VK_MVK_ios_surface . . . 1083
VK_MVK_macos_surface . . . 1084
VK_NN_vi_surface . . . 1086

VK_NV_clip_space_w_scaling . . . 1087
VK_NV_dedicated_allocation . . . 1091
VK_NV_external_memory_capabilities . . . 1094
VK_NV_external_memory . . . 1096
VK_NV_external_memory_win32 . . . 1098
VK_NV_geometry_shader_passthrough . . . 1103
VK_NV_glsl_shader . . . 1107
VK_NV_viewport_array2 . . . 1109
VK_NV_viewport_swizzle . . . 1111
VK_NV_win32_keyed_mutex . . . 1116
VK_NV_sample_mask_override_coverage . . . 1120
VK_NVX_device_generated_commands . . . 1122
VK_NVX_multiview_per_view_attributes . . . 1130
Appendix D: API Boilerplate . . . 1134
Structure Types . . . 1134
Flag Types . . . 1137
Macro Definitions in vulkan.h . . . 1141
Platform-Specific Macro Definitions in vk_platform.h . . . 1144
Appendix E: Invariance . . . 1146
Repeatability . . . 1146
Multi-pass Algorithms . . . 1146
Invariance Rules . . . 1146
Tessellation Invariance . . . 1148
Glossary . . . 1150
Common Abbreviations . . . 1168
Prefixes . . . 1169
Appendix F: Credits . . . 1170

Copyright © 2014-2017 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. 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 Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be reformatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group web-site should be included whenever possible with specification distributions.

This specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at www.khronos.org/files/member_agreement.pdf. 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 do not define requirements necessary for compliance and so are outside the Scope of this Specification. These parts of the Specification are marked by the “Note” icon or designated “Informative”.

Where this Specification uses terms, defined in the Glossary or otherwise, that refer to enabling technologies that are not expressly set forth as being required for compliance, those enabling technologies are outside the Scope of this Specification.

Where this Specification uses the terms “may”, or “optional”, such features or behaviors do not define requirements necessary for compliance and so are outside the Scope of this Specification.

Where this Specification uses the terms “not required”, such features or behaviors may be omitted from certain implementations, but when they are included, they define requirements necessary for compliance and so are INCLUDED in the Scope of this Specification.

Where this Specification includes normative references to external documents, the specifically identified sections and functionality of those external documents are in Scope. Requirements defined by external documents not created by Khronos may contain contributions from non-members of Khronos not covered by the Khronos Intellectual Property Rights Policy.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of

merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, 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.

Khronos is a trademark, and Vulkan is a registered trademark of The Khronos Group Inc. OpenCL is a trademark of Apple Inc. and OpenGL is a registered trademark of Silicon Graphics International, both used under license by Khronos.

# Chapter 1. Introduction

This chapter is Informative except for the sections on Terminology and Normative References.

This document, referred to as the “Vulkan Specification” or just the “Specification” hereafter, describes the Vulkan graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudimentary understanding of computer graphics. This means familiarity with the essentials of computer graphics algorithms and terminology as well as with modern GPUs (Graphic Processing Units).

The canonical version of the Specification is available in the official Vulkan Registry, located at URL

http://www.khronos.org/registry/vulkan/

## 1.1. What is the Vulkan Graphics System?

Vulkan is an API (Application Programming Interface) for graphics and compute hardware. The API consists of many commands that allow a programmer to specify shader programs, compute kernels, objects, and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

### 1.1.1. The Programmer’s View of Vulkan

To the programmer, Vulkan is a set of commands that allow the specification of *shader programs* or *shaders*, *kernels*, data used by kernels or shaders, and state controlling aspects of Vulkan outside of shader execution. Typically, the data represents geometry in two or three dimensions and texture images, while the shaders and kernels control the processing of the data, rasterization of the geometry, and the lighting and shading of *fragments* generated by rasterization, resulting in the rendering of geometry into the framebuffer.

A typical Vulkan program begins with platform-specific calls to open a window or otherwise prepare a display device onto which the program will draw. Then, calls are made to open *queues* to which *command buffers* are submitted. The command buffers contain lists of commands which will be executed by the underlying hardware. The application **can** also allocate device memory, associate *resources* with memory and refer to these resources from within command buffers. Drawing commands cause application-defined shader programs to be invoked, which **can** then consume the data in the resources and use them to produce graphical images. To display the resulting images, further platform-specific commands are made to transfer the resulting image to a display device or window.

### 1.1.2. The Implementor’s View of Vulkan

To the implementor, Vulkan is a set of commands that allow the construction and submission of command buffers to a device. Modern devices accelerate virtually all Vulkan operations, storing data and framebuffer images in high-speed memory and executing shaders in dedicated GPU processing resources.

The implementor’s task is to provide a software library on the host which implements the Vulkan

API, while mapping the work for each Vulkan command to the graphics hardware as appropriate for the capabilities of the device.

### 1.1.3. Our View of Vulkan

We view Vulkan as a pipeline having some programmable stages and some state-driven fixed-function stages that are invoked by a set of specific drawing operations. We expect this model to result in a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation **must** produce results conforming to those produced by the specified methods, but **may** carry out particular computations in ways that are more efficient than the one specified.

## 1.2. Filing Bug Reports

Issues with and bug reports on the Vulkan Specification and the API Registry **can** be filed in the Khronos Vulkan GitHub repository, located at URL

http://github.com/KhronosGroup/Vulkan-Docs

Please tag issues with appropriate labels, such as “Specification”, “Ref Pages” or “Registry”, to help us triage and assign them appropriately. Unfortunately, GitHub does not currently let users who do not have write access to the repository set GitHub labels on issues. In the meantime, they **can** be added to the title line of the issue set in brackets, e.g. "[Specification]".

## 1.3. Terminology

The key words **must**, **required**, **should**, **recommend**, **may**, and **optional** in this document are to be interpreted as described in RFC 2119:

http://www.ietf.org/rfc/rfc2119.txt

**must**

When used alone, this word, or the term **required**, means that the definition is an absolute requirement of the specification. When followed by **not** (“**must** not” ), the phrase means that the definition is an absolute prohibition of the specification.

**should**

When used alone, this word means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course. When followed by **not** (“**should** not”), the phrase means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications **should** be understood and the case carefully weighed before implementing any behavior described with this label. In cases where grammatically appropriate, the terms **recommend** or **recommendation** may be used instead of **should**.

**may**

This word, or the adjective **optional**, means that an item is truly optional. One vendor may

choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option must be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option must be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides).

The additional terms **can** and **cannot** are to be interpreted as follows:

**can**

This word means that the particular behavior described is a valid choice for an application, and is never used to refer to implementation behavior.

**cannot**

This word means that the particular behavior described is not achievable by an application. For example, an entry point does not exist, or shader code is not capable of expressing an operation.

*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.

## 1.4. Normative References

Normative references are references to external documents or resources to which implementers of Vulkan **must** comply.

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

A. Garrard, *Khronos Data Format Specification, version 1.1*, https://www.khronos.org/registry/dataformat/specs/1.1/dataformat.1.1.html, June, 2016.

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

J. Kessenich and B. Ouriel, *The Khronos SPIR-V Specification, Version 1.00*, https://www.khronos.org/registry/spir-v/, February 10, 2016.

J. Leech and T. Hector, *Vulkan Documentation and Extensions: Procedures and Conventions*, https://www.khronos.org/registry/vulkan/, July 11, 2016

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

# 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. Architecture Model

Vulkan is designed for, and the API is written for, CPU, GPU, and other hardware accelerator architectures with the following properties:

- Runtime support for 8, 16, 32 and 64-bit signed and unsigned twos-complement integers, all addressable at the granularity of their size in bytes.
- 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 **must** be identical for the host and the physical devices.

> *Note*
>
> Since a variety of data types and structures in Vulkan **may** be mapped back and forth between host and physical device memory, host and device architectures **must** both be able to access such data efficiently in order to write portable and performant applications.

Where the Specification leaves choices open that would affect Application Binary Interface compatibility on a given platform supporting Vulkan, those choices are usually made to be compliant to the preferred ABI defined by the platform vendor. Some choices, such as function calling conventions, **may** be made in platform-specific portions of the vk_platform.h header file.

> *Note*
>
> For example, the Android ABI is defined by Google, and the Linux ABI is defined by a combination of gcc defaults, distribution vendor choices, and external standards such as the Linux Standard Base.

## 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 hardware 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 pointing to a software structure. 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.

Application-owned memory is immediately consumed by any Vulkan command it is passed into. The application **can** alter or free this memory as soon as the commands that consume 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`

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, descriptor set layouts **can** be destroyed any time they are not in use by an API command.

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`
- `VkDescriptorPool`
- `VkFramebuffer`
- `VkRenderPass`
- `VkCommandBuffer`
- `VkCommandPool`
- `VkDeviceMemory`
- `VkDescriptorSet`
- `VkObjectTableNVX`
- `VkIndirectCommandsLayout`

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`
  - `VkDescriptorSetLayout`
  - `VkDescriptorPool`
  - `VkFramebuffer`
  - `VkRenderPass`
  - `VkCommandPool`
  - `VkCommandBuffer`
  - `VkDeviceMemory`
- `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. 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;
```

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.

### 2.4.1. 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.

## 2.5. 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 software structures representing Vulkan objects. A parameter declared as externally synchronized **may** have its software structures 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 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.

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 such that they do not require external synchronization. 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 `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` 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` 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 `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 `display` parameter in vkCreateDisplayModeKHR
- The `mode` parameter in vkGetDisplayPlaneCapabilitiesKHR
- The `commandPool` parameter in vkTrimCommandPoolKHR
- The `commandBuffer` parameter in vkCmdPushDescriptorSetKHR
- The `descriptorUpdateTemplate` parameter in vkDestroyDescriptorUpdateTemplateKHR
- The `descriptorSet` parameter in vkUpdateDescriptorSetWithTemplateKHR
- The `commandBuffer` parameter in vkCmdPushDescriptorSetWithTemplateKHR
- The `swapchain` parameter in vkGetSwapchainStatusKHR
- 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 vkCmdDrawIndirectCountAMD
- The `commandBuffer` parameter in vkCmdDrawIndexedIndirectCountAMD
- The `commandBuffer` parameter in vkCmdSetDeviceMaskKHX
- The `surface` parameter in vkGetDeviceGroupSurfacePresentModesKHX
- The `commandBuffer` parameter in vkCmdDispatchBaseKHX
- The `surface` parameter in vkGetPhysicalDevicePresentRectanglesKHX
- 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

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

- Each element of the `pWaitSemaphores` member of each element of the `pSubmits` parameter in vkQueueSubmit
- Each element of the `pSignalSemaphores` member of each element of the `pSubmits` parameter in vkQueueSubmit
- Each element of the `pWaitSemaphores` member of each element of the `pBindInfo` parameter in vkQueueBindSparse
- Each element of the `pSignalSemaphores` member of each element of the `pBindInfo` parameter in vkQueueBindSparse
- 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

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 vkCmdPushDescriptorSetKHR
- The VkCommandPool that commandBuffer was allocated from, in vkCmdPushDescriptorSetWithTemplateKHR
- 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 vkCmdDrawIndirectCountAMD
- The VkCommandPool that commandBuffer was allocated from, in vkCmdDrawIndexedIndirectCountAMD
- The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDeviceMaskKHX
- The VkCommandPool that commandBuffer was allocated from, in vkCmdDispatchBaseKHX
- 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

## 2.6. 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, and does not allow one application to access data belonging to another application. Applications **can** request stronger robustness guarantees by enabling the `robustBufferAccess` feature as described in Features, Limits, and Formats.

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.

### 2.6.1. 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.

### 2.6.2. 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. 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 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`.

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 accomodate 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. These aliases are described in the Flag Types appendix of the Specification.

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. As a general rule, the name of this value is obtained by taking the structure name, 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` **must** have a `sType` value of `VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO`.

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.

The list of supported structure types is defined in an appendix.

#### 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 points 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.

#### 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.

### 2.6.3. 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_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_OUT_OF_POOL_MEMORY_KHR = -1000069000,
    VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX = -1000072003,
} 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 accomodate the new allocation. This **should** be returned in preference to `VK_ERROR_OUT_OF_POOL_MEMORY_KHR`, 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_KHR` A pool memory allocation has failed. This **must** only be returned if no attempt to allocate host or device memory was made to accomodate 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_KHX` An external handle is not a valid handle of the specified type.

If a command returns a run time error, it will leave any result pointers unmodified, unless other behavior is explicitly defined in the specification.

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`.

# 2.7. 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.

## 2.7.1. 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.

Any representable floating-point value 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. In IEEE 754 arithmetic, for example, providing a negative zero or a denormalized number to an Vulkan command **must** yield deterministic results, while providing a NaN or Inf yields unspecified results.

### 2.7.2. 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.

Any representable 16-bit floating-point value is legal as input to a Vulkan command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but **must** not lead to Vulkan interruption or termination. Providing a denormalized number or negative zero to Vulkan **must** yield deterministic results.

### 2.7.3. 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.

When a floating-point value is converted to an unsigned 11-bit floating-point representation, finite values are rounded to the closest representable finite value.

While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 65024 (the maximum finite representable unsigned 11-bit floating-point value) are converted to 65024. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 11-bit floating-point value is legal as input to a Vulkan command that accepts 11-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but **must** not lead to Vulkan interruption or termination. Providing a denormalized number to Vulkan **must** yield deterministic results.

### 2.7.4. 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.

When a floating-point value is converted to an unsigned 10-bit floating-point representation, finite values are rounded to the closest representable finite value.

While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 64512 (the maximum finite representable unsigned 10-bit floating-point value) are converted to 64512. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 10-bit floating-point value is legal as input to a Vulkan command that accepts 10-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but **must** not lead to Vulkan interruption or termination. Providing a denormalized number to Vulkan **must** yield deterministic results.

### 2.7.5. General Requirements

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.

## 2.8. 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.

### 2.8.1. 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.

### 2.8.2. 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

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

where convertFloatToUint}(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, 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.

## 2.9. API Version Numbers and Semantics

The Vulkan version number is used in several places in the API. In each such use, the API *major version number*, *minor version number*, and *patch version number* are packed into a 32-bit integer as follows:

- The major version number 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.

Differences in any of the Vulkan version numbers indicates a change to the API in some way, with each part of the version number indicating a different scope of changes.

A difference in patch version numbers indicates that some usually small part of the specification or header has been modified, typically to fix a bug, and **may** have an impact on the behavior of existing functionality. Differences in this version number **should** not affect either *full compatibility* or *backwards compatibility* between two versions, or add additional interfaces to the API.

A difference in minor version numbers indicates that some amount of new functionality has been added. This will usually include new interfaces in the header, and **may** also include behavior changes and bug fixes. Functionality **may** be deprecated in a minor revision, but will not be removed. When a new minor version is introduced, the patch version is reset to 0, and each minor revision maintains its own set of patch versions. Differences in this version **should** not affect backwards compatibility, but will affect full compatibility.

A difference in major version numbers indicates a large set of changes to the API, potentially including new functionality and header interfaces, behavioral changes, removal of deprecated features, modification or outright replacement of any feature, and is thus very likely to break any and all compatibility. Differences in this version will typically require significant modification to an application in order for it to function.

C language macros for manipulating version numbers are defined in the Version Number Macros

appendix.

## 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 offsets is defined by the structure:

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

A three-dimensional offset is defined by the structure:

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

### 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;
```

A three-dimensional extent is defined by the structure:

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

### 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;
```

# 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 statically on a platform. 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. Loaders are encouraged to export function symbols for all other core Vulkan commands as well; if this is done, then applications that use only the core Vulkan commands have no need to use `vkGetInstanceProcAddr`.

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*

| `instance` | `pName` | **return value** |
|---|---|---|
| * | `NULL` | undefined |
| invalid instance | * | undefined |
| `NULL` | vkEnumerateInstanceExtensionProperties | fp |
| `NULL` | vkEnumerateInstanceLayerProperties | fp |
| `NULL` | vkCreateInstance | fp |
| `NULL` | * (any `pName` not covered above) | `NULL` |
| instance | core Vulkan command | fp[1] |

| instance | pName | return value |
|---|---|---|
| instance | enabled instance extension commands for instance | fp[1] |
| instance | available device extension[2] commands for instance | fp[1] |
| instance | * (any pName not covered above) | NULL |

**1**

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.

**2**

An "available extension" is an extension function supported by any of the loader, driver or layer.

### Valid Usage (Implicit)

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

In order to support systems with multiple Vulkan implementations comprising heterogeneous collections of hardware and software, the function pointers returned by vkGetInstanceProcAddr **may** point to dispatch code, which calls a different real implementation for different VkDevice objects (and objects created from them). The overhead of this internal dispatch **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.

*Table 2. vkGetDeviceProcAddr behavior*

| device | pName | return value |
|---|---|---|
| NULL | * | undefined |
| invalid device | * | undefined |

| device | pName | return value |
|---|---|---|
| device | NULL | undefined |
| device | core Vulkan command | fp[1] |
| device | enabled extension commands | fp[1] |
| device | * (any pName not covered above) | NULL |

**1**

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 string

The definition of PFN_vkVoidFunction is:

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

### 3.1.1. Extending Physical Device From Device Extensions

When the VK_KHR_get_physical_device_properties2 extension is enabled, physical device extension commands and structures **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 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. Behavior is undefined if an extension physical-device command is called on a physical device that does not support the extension.

Device extensions **may** define structures that **can** be added to the pNext chain of physical-device commands. Behavior is undefined if such an extension structure is passed to a physical device command for a physical device that does not support the extension.

## 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 create an instance object, call:

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

- `pCreateInfo` points to an instance of VkInstanceCreateInfo 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 (Implicit)

- `pCreateInfo` **must** be a pointer to a valid `VkInstanceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pInstance` **must** be a 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 an instance of `VkApplicationInfo`. 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`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkDebugReportCallbackCreateInfoEXT
- `flags` **must** be `0`
- If `pApplicationInfo` is not `NULL`, `pApplicationInfo` **must** be a pointer to a valid `VkApplicationInfo` structure
- If `enabledLayerCount` is not `0`, `ppEnabledLayerNames` **must** be a pointer to an array of `enabledLayerCount` null-terminated strings
- If `enabledExtensionCount` is not `0`, `ppEnabledExtensionNames` **must** be a pointer to an array of `enabledExtensionCount` null-terminated strings

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;
    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 values specifying the validation checks to be disabled. Checks which **may** be specified include:

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

  - `VK_VALIDATION_CHECK_ALL_EXT` disables all validation checks.
  - `VK_VALIDATION_CHECK_SHADERS_EXT` disables all shader validation.

### Valid Usage (Implicit)

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

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 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 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 is the version of the Vulkan API against which the application expects to run, encoded as described in the API Version Numbers and Semantics section. If apiVersion is 0 the implementation **must** ignore it, otherwise if the implementation does not support the requested apiVersion, or an effective substitute for apiVersion, it **must** return VK_ERROR_INCOMPATIBLE_DRIVER. 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.

### 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 string
- If pEngineName is not NULL, pEngineName **must** be a null-terminated string

To destroy an instance, call:

```
void vkDestroyInstance(
    VkInstance                                  instance,
    const VkAllocationCallbacks*                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 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 device in a system (perhaps made up of several individual hardware devices working together), of which there are a finite number. A logical device represents an application's view of the device.

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                                  instance,
    uint32_t*                                   pPhysicalDeviceCount,
    VkPhysicalDevice*                           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 pointer to a `uint32_t` value
- If the value referenced by `pPhysicalDeviceCount` is not `0`, and `pPhysicalDevices` is not `NULL`, `pPhysicalDevices` **must** be a 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` points to an instance of the VkPhysicalDeviceProperties structure, that will be filled with returned information.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pProperties` **must** be a 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 the API Version Numbers and Semantics section.
- `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 a null-terminated UTF-8 string containing the name of the device.
- `pipelineCacheUUID` is an array of size `VK_UUID_SIZE`, containing 8-bit values that represent a universally unique identifier for the device.
- `limits` is the VkPhysicalDeviceLimits structure which specifies device-specific limits of the physical device. See Limits for details.
- `sparseProperties` is the VkPhysicalDeviceSparseProperties structure which specifies various sparse related properties of the physical device. See Sparse Properties for details.

The `vendorID` and `deviceID` fields are provided to allow applications to adapt to device characteristics that are not adequately exposed by other Vulkan queries. These **may** include performance profiles, hardware errata, or other characteristics. In PCI-based implementations, the low sixteen bits of `vendorID` and `deviceID` **must** contain (respectively) the PCI vendor and device IDs associated with the hardware device, and the remaining bits **must** be set to zero. In non-PCI implementations, the choice of what values to return **may** be dictated by operating system or platform policies. It is otherwise at the discretion of the implementer, subject to the following constraints and guidelines:

- For purposes of physical device identification, the *vendor* of a physical device is the entity responsible for the most salient characteristics of the hardware represented by the physical device handle. In the case of a discrete GPU, this **should** be the GPU chipset vendor. In the case of a GPU or other accelerator integrated into a system-on-chip (SoC), this **should** be the supplier of the silicon IP used to create the GPU or other accelerator.
- If the vendor of the physical device has a valid PCI vendor ID issued by PCI-SIG, that ID **should** be used to construct `vendorID` as described above for PCI-based implementations. Implementations that do not return a PCI vendor ID in `vendorID` **must** return a valid Khronos vendor ID, obtained as described in the Vulkan Documentation and Extensions 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.

- The vendor of the physical device is responsible for selecting `deviceID`. The value selected **should** uniquely identify both the device version and any major configuration options (for example, core count in the case of multicore devices). 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.

The physical devices types 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,
} 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 vkGetPhysicalDeviceProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceProperties2KHR*             pProperties);
```

- `physicalDevice` is the handle to the physical device whose properties will be queried.
- `pProperties` points to an instance of the VkPhysicalDeviceProperties2KHR structure, that will be filled with returned information.

Each structure in `pProperties` and its `pNext` chain contain members corresponding to properties or implementation-dependent limits. `vkGetPhysicalDeviceProperties2KHR` 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 pointer to a `VkPhysicalDeviceProperties2KHR` structure

The `VkPhysicalDeviceProperties2KHR` structure is defined as:

```
typedef struct VkPhysicalDeviceProperties2KHR {
    VkStructureType                 sType;
    void*                           pNext;
    VkPhysicalDeviceProperties      properties;
} VkPhysicalDeviceProperties2KHR;
```

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

To query the UUID and LUID of a device, add VkPhysicalDeviceIDPropertiesKHX to the `pNext` chain of the VkPhysicalDeviceProperties2KHR structure. The `VkPhysicalDeviceIDPropertiesKHX` structure is defined as:

```
typedef struct VkPhysicalDeviceIDPropertiesKHX {
    VkStructureType     sType;
    void*               pNext;
    uint8_t             deviceUUID[VK_UUID_SIZE];
    uint8_t             driverUUID[VK_UUID_SIZE];
    uint8_t             deviceLUID[VK_LUID_SIZE_KHX];
    VkBool32            deviceLUIDValid;
} VkPhysicalDeviceIDPropertiesKHX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `deviceUUID` is an array of size `VK_UUID_SIZE`, containing 8-bit values that represent a universally unique identifier for the device.
- `driverUUID` is an array of size `VK_UUID_SIZE`, containing 8-bit values that represent a universally unique identifier for the driver build in use by the device.
- `deviceLUID` is an array of size `VK_LUID_SIZE_KHX`, containing 8-bit values that represent a locally unique identifier for the device.

- `deviceLUIDValid` is a boolean value that will be `VK_TRUE` if `deviceLUID` contains a valid LUID, and `VK_FALSE` if it does not.

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

`driverUUID` **must** be identical in all driver components that support sharing external objects between eachother. Applications **can** compare this 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.

If `deviceLUIDValid` is `VK_FALSE`, the contents of `deviceLUID` 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`.

> *Note*
>
> Although they have identical descriptions, VkPhysicalDeviceIDPropertiesKHX::`deviceUUID` may differ from VkPhysicalDeviceProperties2KHR::`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.

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`. 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 pointer to a `uint32_t` value
- If the value referenced by `pQueueFamilyPropertyCount` is not `0`, and `pQueueFamilyProperties` is not `NULL`, `pQueueFamilyProperties` **must** be a 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` contains flags indicating the capabilities of the queues in this queue family.
- `queueCount` is the unsigned integer count of queues in this queue family.
- `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 bits specified in `queueFlags` 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,
} VkQueueFlagBits;
```

- if `VK_QUEUE_GRAPHICS_BIT` is set, then the queues in this queue family support graphics operations.
- if `VK_QUEUE_COMPUTE_BIT` is set, then the queues in this queue family support compute operations.
- if `VK_QUEUE_TRANSFER_BIT` is set, then the queues in this queue family support transfer operations.
- if `VK_QUEUE_SPARSE_BINDING_BIT` is set, then the 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 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.

> *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.

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.

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

```
void vkGetPhysicalDeviceQueueFamilyProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t*                                   pQueueFamilyPropertyCount,
    VkQueueFamilyProperties2KHR*                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 VkQueueFamilyProperties2KHR structures.

`vkGetPhysicalDeviceQueueFamilyProperties2KHR` 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 pointer to a `uint32_t` value
- If the value referenced by `pQueueFamilyPropertyCount` is not `0`, and `pQueueFamilyProperties` is not `NULL`, `pQueueFamilyProperties` **must** be a pointer to an array of `pQueueFamilyPropertyCount` `VkQueueFamilyProperties2KHR` structures

The `VkQueueFamilyProperties2KHR` structure is defined as:

```
typedef struct VkQueueFamilyProperties2KHR {
    VkStructureType             sType;
    void*                       pNext;
    VkQueueFamilyProperties     queueFamilyProperties;
} VkQueueFamilyProperties2KHR;
```

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

## 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 vkEnumeratePhysicalDeviceGroupsKHX, and a logical device is created from a subset of the physical devices in a device group by passing the physical devices through VkDeviceGroupDeviceCreateInfoKHX.

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

```
VkResult vkEnumeratePhysicalDeviceGroupsKHX(
    VkInstance                                  instance,
    uint32_t*                                   pPhysicalDeviceGroupCount,
    VkPhysicalDeviceGroupPropertiesKHX*         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 VkPhysicalDeviceGroupPropertiesKHX 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 pointer to a uint32_t value
- If the value referenced by pPhysicalDeviceGroupCount is not 0, and pPhysicalDeviceGroupProperties is not NULL, pPhysicalDeviceGroupProperties **must** be a pointer to an array of pPhysicalDeviceGroupCount VkPhysicalDeviceGroupPropertiesKHX 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 VkPhysicalDeviceGroupPropertiesKHX structure is defined as:

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

- 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 physical device handles representing all physical devices in the group. The first physicalDeviceCount elements of the array will be valid.
- subsetAllocation indicates whether logical devices created from the group support allocating device memory on a subset of devices, via the deviceMask member of the VkMemoryAllocateFlagsInfoKHX. 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.

### 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` points to a handle in which the created `VkDevice` is returned.

`vkCreateDevice` 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, `vkCreateDevice` **must** return `VK_ERROR_EXTENSION_NOT_PRESENT`. 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 (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pCreateInfo` **must** be a pointer to a valid `VkDeviceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pDevice` **must** be a 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 that contains boolean indicators of all the features to be enabled. Refer to the Features section for further details.

### Valid Usage

- The queueFamilyIndex member of any given element of pQueueCreateInfos **must** be unique within pQueueCreateInfos
- If the pNext chain includes a VkPhysicalDeviceFeatures2KHR structure, then pEnabledFeatures **must** be NULL
- ppEnabledExtensionNames **must** not contain both VK_KHR_maintenance1 and VK_AMD_negative_viewport_height

### 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 VkPhysicalDeviceFeatures2KHR, VkPhysicalDeviceMultiviewFeaturesKHX, or VkDeviceGroupDeviceCreateInfoKHX
- Each sType member in the pNext chain **must** be unique
- flags **must** be 0
- pQueueCreateInfos **must** be a pointer to an array of queueCreateInfoCount valid VkDeviceQueueCreateInfo structures
- If enabledLayerCount is not 0, ppEnabledLayerNames **must** be a pointer to an array of enabledLayerCount null-terminated strings
- If enabledExtensionCount is not 0, ppEnabledExtensionNames **must** be a pointer to an array of enabledExtensionCount null-terminated strings
- If pEnabledFeatures is not NULL, pEnabledFeatures **must** be a pointer to a valid VkPhysicalDeviceFeatures structure
- queueCreateInfoCount **must** be greater than 0

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

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

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

## 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* because of hardware errors, execution timeouts, power management events and/or platform-specific events. This **may** cause pending and future command execution to fail and cause hardware resources to be corrupted. 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 hardware, 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 their hardware has probably developed a fault or become physically disconnected, and **should** be investigated further. 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.

*Note*

Undefined behavior caused by an application error **may** cause a device to become lost. However, such undefined behavior **may** also cause unrecoverable damage to the process, and it is then not guaranteed that the API objects, including the `VkPhysicalDevice` or the `VkInstance` are still valid or that the error is recoverable.

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.

### 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 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 reserved for future use.
- 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 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
- flags **must** be 0
- pQueuePriorities **must** be a pointer to an array of queueCount float values
- queueCount **must** be greater than 0

To retrieve a handle to a VkQueue object, call:

```
void vkGetDeviceQueue(
    VkDevice                                    device,
    uint32_t                                    queueFamilyIndex,
    uint32_t                                    queueIndex,
    VkQueue*                                    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.

### 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

### Valid Usage (Implicit)

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

## 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 Events) 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. Whenever the state of a command buffer is undefined, the application **must** set all relevant state on the command buffer before any state dependent commands such as draws and dispatches are recorded, otherwise the behavior of executing that command buffer is 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 first allocated 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 - the device **may** be processing the commands recorded to it. Once execution of a command buffer completes, the command buffer reverts back to the executable state. 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 a command buffer into the *invalid state*. Command buffers in the invalid state **can** only be reset, moved to the *recording state*, or freed.

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` contains information used to create the command pool.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pCommandPool` points to a `VkCommandPool` handle in which the created pool is returned.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a pointer to a valid `VkCommandPoolCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pCommandPool` **must** be a 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 indicating usage behavior for the pool and command buffers allocated from it. Bits which **can** be set include:

  ```
  typedef enum VkCommandPoolCreateFlagBits {
      VK_COMMAND_POOL_CREATE_TRANSIENT_BIT = 0x00000001,
      VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT = 0x00000002,
  } VkCommandPoolCreateFlagBits;
  ```

  - `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` indicates 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.

- `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

- `queueFamilyIndex` **must** be the index of a queue family available in the calling command's `device` parameter

### 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

To trim a command pool, call:

```
void vkTrimCommandPoolKHR(
    VkDevice                                    device,
    VkCommandPool                               commandPool,
    VkCommandPoolTrimFlagsKHR                   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" behaviour.

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

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` contains additional flags controlling the behavior of the reset. Bits which **can** be set include:

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

  If `flags` includes `VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT`, resetting a command pool recycles all of the resources from the command pool back to the system.

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`

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 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 an instance of the `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 pointer to a valid `VkCommandBufferAllocateInfo` structure
- `pCommandBuffers` **must** be a pointer to an array of `pAllocateInfo`::commandBufferCount `VkCommandBuffer` handles

### 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` determines whether the command buffers are primary or secondary command buffers. Possible values include:

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

- `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

To reset command buffers, call:

```
VkResult vkResetCommandBuffer(
    VkCommandBuffer                             commandBuffer,
    VkCommandBufferResetFlags                   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 controlling the reset operation. Bits which **can** be set include:

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

  If `flags` includes `VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT`, then 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.

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`

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 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 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                             commandBuffer,
    const VkCommandBufferBeginInfo*             pBeginInfo);
```

- `commandBuffer` is the handle of the command buffer which is to be put in the recording state.
- `pBeginInfo` is an instance of the `VkCommandBufferBeginInfo` structure, which defines 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 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 indicating usage behavior for the command buffer. Bits which **can** be set include:

```
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,
} VkCommandBufferUsageFlagBits;
```

  - `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` indicates 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` indicates 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.
  - Setting `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` allows the command buffer to be resubmitted to a queue while it is in the *pending state*, and recorded into multiple primary command buffers.
- `pInheritanceInfo` is a pointer to a `VkCommandBufferInheritanceInfo` structure, which is 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 VkDeviceGroupCommandBufferBeginInfoKHX
- `flags` **must** be a valid combination of VkCommandBufferUsageFlagBits values

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 indicates 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 indicates 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 indicates 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 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`
- Both of `framebuffer`, and `renderPass` that are valid handles **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.

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
- 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. 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 **can** not 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.
- Any given 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 unsignal operation defined by any element of the `pWaitSemaphores` member of any element of `pSubmits` executes on `queue`, no other queue **must** be 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.
- Any given element of the `pCommandBuffers` member of any element of `pSubmits` **must** be in the pending or executable state.
- If any given 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 given 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 given 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 given element of the `pCommandBuffers` member of any element of `pSubmits` **must** have been allocated from a `VkCommandPool` that was created for the same queue family `queue` belongs to.

## Valid Usage (Implicit)

- `queue` **must** be a valid `VkQueue` handle
- If `submitCount` is not `0`, `pSubmits` **must** be a 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 **must** have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to `queue` **must** be externally synchronized
- Host access to `pSubmits`[].pWaitSemaphores[] **must** be externally synchronized
- Host access to `pSubmits`[].pSignalSemaphores[] **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 semaphores 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 command buffers 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 semaphores 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

- Any given element of `pCommandBuffers` **must** not have been allocated with `VK_COMMAND_BUFFER_LEVEL_SECONDARY`
- If the geometry shaders feature is not enabled, any given element of `pWaitDstStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the tessellation shaders feature is not enabled, any given element of `pWaitDstStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- Any given element of `pWaitDstStageMask` **must** not include `VK_PIPELINE_STAGE_HOST_BIT`.

### 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 VkWin32KeyedMutexAcquireReleaseInfoNV, VkWin32KeyedMutexAcquireReleaseInfoKHX, VkD3D12FenceSubmitInfoKHX, or VkDeviceGroupSubmitInfoKHX
- Each `sType` member in the `pNext` chain **must** be unique
- If `waitSemaphoreCount` is not `0`, `pWaitSemaphores` **must** be a pointer to an array of `waitSemaphoreCount` valid `VkSemaphore` handles
- If `waitSemaphoreCount` is not `0`, `pWaitDstStageMask` **must** be a 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 pointer to an array of `commandBufferCount` valid `VkCommandBuffer` handles
- If `signalSemaphoreCount` is not `0`, `pSignalSemaphores` **must** be a 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 **must** have been created, allocated, or retrieved from the same `VkDevice`

To specify the values to use when waiting for and signaling semaphores whose current state is shared with a Direct3D 12 fence, add the VkD3D12FenceSubmitInfoKHX structure to the `pNext` chain of the VkSubmitInfo structure. The `VkD3D12FenceSubmitInfoKHX` structure is defined as:

```
typedef struct VkD3D12FenceSubmitInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           waitSemaphoreValuesCount;
    const uint64_t*    pWaitSemaphoreValues;
    uint32_t           signalSemaphoreValuesCount;
    const uint64_t*    pSignalSemaphoreValues;
} VkD3D12FenceSubmitInfoKHX;
```

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

If the semaphore in VkSubmitInfo::`pWaitSemaphores` or VkSubmitInfo::`pSignalSemaphore` corresponding to an entry in `pWaitSemaphoreValues` or `pSignalSemaphoreValues` respectively does not currently have state imported from a Direct3D 12 fence, the value in the `pWaitSemaphoreValues` or `pSignalSemaphoreValues` entry is ignored.

### Valid Usage

- `waitSemaphoreValuesCount` **must** be the same value as `VkSubmitInfo`::`waitSemaphoreCount`, where `SubmitInfo` is in the `pNext` chain of this `VkD3D12FenceSubmitInfoKHX` structure.
- `signalSemaphoreValuesCount` **must** be the same value as `VkSubmitInfo`::`signalSemaphoreCount`, where `SubmitInfo` is in the `pNext` chain of this `VkD3D12FenceSubmitInfoKHX` structure.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHX`
- If `waitSemaphoreValuesCount` is not `0`, and `pWaitSemaphoreValues` is not `NULL`, `pWaitSemaphoreValues` **must** be a pointer to an array of `waitSemaphoreValuesCount` `uint64_t` values
- If `signalSemaphoreValuesCount` is not `0`, and `pSignalSemaphoreValues` is not `NULL`, `pSignalSemaphoreValues` **must** be a 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 VkWin32KeyedMutexAcquireReleaseInfoKHX structure to the `pNext` chain of the VkSubmitInfo structure.

The `VkWin32KeyedMutexAcquireReleaseInfoKHX` structure is defined as:

```
typedef struct VkWin32KeyedMutexAcquireReleaseInfoKHX {
    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;
} VkWin32KeyedMutexAcquireReleaseInfoKHX;
```

- `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 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 VkImportMemoryWin32HandleInfoKHX::`handleType` to `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHX` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT_KHX`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHX`
- If `acquireCount` is not `0`, `pAcquireSyncs` **must** be a pointer to an array of `acquireCount` valid `VkDeviceMemory` handles
- If `acquireCount` is not `0`, `pAcquireKeys` **must** be a pointer to an array of `acquireCount` `uint64_t` values
- If `acquireCount` is not `0`, `pAcquireTimeouts` **must** be a pointer to an array of `acquireCount` `uint32_t` values
- If `releaseCount` is not `0`, `pReleaseSyncs` **must** be a pointer to an array of `releaseCount` valid `VkDeviceMemory` handles
- If `releaseCount` is not `0`, `pReleaseKeys` **must** be a pointer to an array of `releaseCount` `uint64_t` values
- Both of the elements of `pAcquireSyncs`, and the elements of `pReleaseSyncs` that are valid handles **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 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 pointer to an array of acquireCount valid VkDeviceMemory handles
- If acquireCount is not 0, pAcquireKeys **must** be a pointer to an array of acquireCount uint64_t values
- If acquireCount is not 0, pAcquireTimeoutMilliseconds **must** be a pointer to an array of acquireCount uint32_t values
- If releaseCount is not 0, pReleaseSyncs **must** be a pointer to an array of releaseCount valid VkDeviceMemory handles
- If releaseCount is not 0, pReleaseKeys **must** be a pointer to an array of releaseCount uint64_t values
- Both of the elements of pAcquireSyncs, and the elements of pReleaseSyncs that are valid handles **must** have been created, allocated, or retrieved from the same VkDevice

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

The VkDeviceGroupSubmitInfoKHX structure is defined as:

```
typedef struct VkDeviceGroupSubmitInfoKHX {
    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;
} VkDeviceGroupSubmitInfoKHX;
```

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

## 5.6. Queue Forward Progress

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`, for every queued wait on a semaphore there **must** be a prior signal of that semaphore that will not be consumed by a different wait on the semaphore.

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. Implementations **may** have limits on how long the command buffer will wait, in order to avoid interfering with progress of other clients of the device. If the event is not signaled within these limits, results are undefined and **may** include device loss.

## 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                                 commandBuffer,
    uint32_t                                        commandBufferCount,
    const VkCommandBuffer*                          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 an array of 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`
- Any given element of `pCommandBuffers` **must** have been allocated with a `level` of `VK_COMMAND_BUFFER_LEVEL_SECONDARY`
- Any given 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 given 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 given 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 given 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`.
- Any given 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, any given 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, any given 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 pname::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 given 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, any given 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
- Any given element of `pCommandBuffers` **must** not begin any query types that are active in `commandBuffer`

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pCommandBuffers` **must** be a 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<br>graphics<br>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 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 VkDeviceGroupSubmitInfoKHX. Commands only execute on a physical device if the device index is set in both device masks.

If the `pNext` list of VkCommandBufferBeginInfo includes a `VkDeviceGroupCommandBufferBeginInfoKHX` structure, then that structure includes an initial device mask for the command buffer.

The `VkDeviceGroupCommandBufferBeginInfoKHX` structure is defined as:

```
typedef struct VkDeviceGroupCommandBufferBeginInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           deviceMask;
} VkDeviceGroupCommandBufferBeginInfoKHX;
```

- `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_KHX`

To update the current device mask of a command buffer, call:

```
void vkCmdSetDeviceMaskKHX(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    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.

### 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 VkDeviceGroupCommandBufferBeginInfoKHX::`deviceMask` value when the command buffer began recording.
- If `vkCmdSetDeviceMaskKHX` is called inside a render pass instance, `deviceMask` **must** not include any set bits that were not in the VkDeviceGroupRenderPassBeginInfoKHX::`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<br>Secondary | Both | Graphics<br>compute<br>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, four explicit synchronization primitives 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.

In addition to the base primitives provided here, 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 synchronization primitives in this chapter **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 $\mathbf{A_S}$ and $\mathbf{B_S}$ be the synchronization scopes of **S**.
- Let **A'** be the intersection of sets **A** and $\mathbf{A_S}$.
- Let **B'** be the intersection of sets **B** and $\mathbf{B_S}$.
- 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 $\mathbf{B_S}$ in the first dependency and $\mathbf{A_S}$ 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 $\mathbf{A_S}$ be the first synchronization scope of the first command in **S**.
- Let $\mathbf{B_S}$ be the second synchronization scope of the last command in **S**.

> *Note*
>
> An execution dependency is inherently also multiple execution dependencies - a dependency exists between each subset of **A'** and each subset of **B'**, and the same is true for execution dependency chains. For example, a synchronization command with multiple pipeline stages in its stage masks effectively generates one dependency between each source stage and each destination stage. This can be useful to think about when considering how execution chains are formed if they do not involve all parts of a synchronization command's dependency. Similarly, any set of adjacent dependencies in an execution dependency chain **can** be considered an execution dependency chain in its own right.

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.

Two additional types of operation are used to control memory access. *Availability operations* cause the values generated by specified memory write accesses to become *available* 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. *Visibility operations* cause any available values to become *visible* to specified memory accesses.

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 $\mathbf{a}_s$ be the first access scope of the first command in **S**.
- Let $\mathbf{b}_s$ be the second access scope of the last command in **S**.
- Let **a'** be the intersection of sets **a** and $\mathbf{a}_s$.
- Let **b'** be the intersection of sets **b** and $\mathbf{b}_s$.
- 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.

> *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.

The contents of any portion of another resource which aliases memory that is bound to the transitioned image subresource range are undefined after an image layout transition.

### 6.1.2. Pipeline Stages

The work performed by an action command consists of multiple operations, which are performed by a sequence of logically independent execution units known as *pipeline stages*. The exact pipeline stages executed depend on the particular action command that is used, and current command buffer state when the action command was recorded. Drawing commands, dispatching commands, copy commands, and clear commands all execute in different sets of 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.

Pipeline stages are specified using a bitmask:

```
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_COMMAND_PROCESS_BIT_NVX = 0x00020000,
} VkPipelineStageFlagBits;
```

The meaning of each bit is:

- `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`: Stage of the pipeline where any commands are initially received by the queue.
- `VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX`: Stage of the pipeline where device-side generation of commands via vkCmdProcessCommandsNVX is handled.
- `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`: Stage of the pipeline where Draw/DispatchIndirect data structures are consumed. This stage also includes reading commands written by vkCmdProcessCommandsNVX.
- `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT`: Stage of the pipeline where vertex and index buffers are consumed.
- `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`: Vertex shader stage.
- `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`: Tessellation control shader stage.
- `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`: Tessellation evaluation shader stage.
- `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`: Geometry shader stage.
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`: Fragment shader stage.
- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`: 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`: 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`: 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 format.

- `VK_PIPELINE_STAGE_TRANSFER_BIT`: 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_COMPUTE_SHADER_BIT`: Execution of a compute shader.
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`: Final stage in the pipeline where operations generated by all commands complete execution.
- `VK_PIPELINE_STAGE_HOST_BIT`: 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_ALL_GRAPHICS_BIT`: Execution of all graphics pipeline stages. Equivalent to the logical or of:
  - `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_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_ALL_COMMANDS_BIT`: Equivalent to the logical or of every other pipeline stage flag that is supported on the queue it is used with.

*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.

If a synchronization command includes a source stage mask, its first synchronization scope only includes execution of the pipeline stages specified in that mask, as well as any logically earlier stages. If a synchronization command includes a destination stage mask, its second synchronization scope only includes execution of the pipeline stages specified in that mask, as well as any logically later stages.

Access scopes are affected in a similar way. If a synchronization command includes a source stage mask, 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 access scope only includes memory access performed by pipeline stages specified in that mask.

*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, then it **may** substitute that stage for any logically later stage. If a pipeline stage that an implementation does not support synchronization for appears in a destination stage mask, then it **may** substitute that stage for any logically earlier stage.

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.

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 |
|---|---|
| VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT | None required |
| VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT | VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT |
| VK_PIPELINE_STAGE_VERTEX_INPUT_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT | VK_QUEUE_COMPUTE_BIT |
| VK_PIPELINE_STAGE_TRANSFER_BIT | VK_QUEUE_GRAPHICS_BIT, VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT |
| VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT | None required |
| VK_PIPELINE_STAGE_HOST_BIT | None required |
| VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT | VK_QUEUE_GRAPHICS_BIT |
| VK_PIPELINE_STAGE_ALL_COMMANDS_BIT | None required |
| VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX | VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT |

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 given stage in the source stage mask for a particular synchronization command also implies that any logically earlier stages are included in $\mathbf{A}_s$ 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 $\mathbf{B}_s$ for that command.

*Note*

Logically earlier/later stages are not included when defining the access scopes of a memory barrier.

The order of pipeline stages depends on the particular pipeline; graphics, compute, transfer or host.

For the graphics 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_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 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

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

### 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_COMMAND_PROCESS_READ_BIT_NVX = 0x00020000,
    VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX = 0x00040000,
} VkAccessFlagBits;
```

- `VK_ACCESS_INDIRECT_COMMAND_READ_BIT`: Read access to an indirect command structure read as part of an indirect drawing or dispatch command.
- `VK_ACCESS_INDEX_READ_BIT`: Read access to an index buffer as part of an indexed drawing command, bound by vkCmdBindIndexBuffer.
- `VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT`: Read access to a vertex buffer as part of a drawing command, bound by vkCmdBindVertexBuffers.
- `VK_ACCESS_UNIFORM_READ_BIT`: Read access to a uniform buffer.
- `VK_ACCESS_INPUT_ATTACHMENT_READ_BIT`: Read access to an input attachment within a renderpass during fragment shading.
- `VK_ACCESS_SHADER_READ_BIT`: Read access to a storage buffer, uniform texel buffer, storage texel buffer, sampled image, or storage image.
- `VK_ACCESS_SHADER_WRITE_BIT`: Write access to a storage buffer, storage texel buffer, or storage image.
- `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`: Read access to a color attachment, such as via blending, logic operations, or via certain subpass load operations.
- `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`: Write access to a color or resolve attachment during a render pass or via certain subpass load and store operations.
- `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT`: Read access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load operations.
- `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`: 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`: Read access to an image or buffer in a copy operation.

- VK_ACCESS_TRANSFER_WRITE_BIT: Write access to an image or buffer in a clear or copy operation.
- VK_ACCESS_HOST_READ_BIT: 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: 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: Read access via non-specific entities. These entities include the Vulkan device and host, but **may** also include entities external to the Vulkan device or otherwise not part of the core Vulkan pipeline. When included in a destination access mask, makes all available writes visible to all future read accesses on entities known to the Vulkan device.
- VK_ACCESS_MEMORY_WRITE_BIT: Write access via non-specific entities. These entities include the Vulkan device and host, but **may** also include entities external to the Vulkan device or otherwise not part of the core Vulkan pipeline. When included in a source access mask, all writes that are performed by entities known to the Vulkan device are made available. When included in a destination access mask, makes all available writes visible to all future write accesses on entities known to the Vulkan device.
- VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX: Reads from VkBuffer inputs to vkCmdProcessCommandsNVX.
- VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX: Writes to the target command buffer in vkCmdProcessCommandsNVX.

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 |
|---|---|
| VK_ACCESS_INDIRECT_COMMAND_READ_BIT | VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT |
| VK_ACCESS_INDEX_READ_BIT | VK_PIPELINE_STAGE_VERTEX_INPUT_BIT |
| VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT | VK_PIPELINE_STAGE_VERTEX_INPUT_BIT |
| VK_ACCESS_UNIFORM_READ_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, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT |
| VK_ACCESS_INPUT_ATTACHMENT_READ_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT |

| Access flag | Supported pipeline stages |
| --- | --- |
| VK_ACCESS_SHADER_READ_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, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT |
| VK_ACCESS_SHADER_WRITE_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, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT |
| VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
| VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
| VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT |
| VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT |
| VK_ACCESS_TRANSFER_READ_BIT | VK_PIPELINE_STAGE_TRANSFER_BIT |
| VK_ACCESS_TRANSFER_WRITE_BIT | VK_PIPELINE_STAGE_TRANSFER_BIT |
| VK_ACCESS_HOST_READ_BIT | VK_PIPELINE_STAGE_HOST_BIT |
| VK_ACCESS_HOST_WRITE_BIT | VK_PIPELINE_STAGE_HOST_BIT |
| VK_ACCESS_MEMORY_READ_BIT | N/A |
| VK_ACCESS_MEMORY_WRITE_BIT | N/A |
| VK_ACCESS_COMMAND_PROCESS_READ_BIT_NVX | VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX |
| VK_ACCESS_COMMAND_PROCESS_WRITE_BIT_NVX | VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX |

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 visible to the VK_ACCESS_HOST_WRITE_BIT access type, where it **can** be further made available to the device by synchronization commands. Similarly, vkInvalidateMappedMemoryRanges **must** be called to guarantee that writes which are visible to the VK_ACCESS_HOST_READ_BIT access type 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 visible to the VK_ACCESS_HOST_WRITE_BIT access type. Similarly, writes made visible to the VK_ACCESS_HOST_READ_BIT access type are automatically made visible to the host.

*Note*

The vkQueueSubmit command automatically guarantees that host writes flushed to `VK_ACCESS_HOST_WRITE_BIT` are made available if they were flushed before the command executed, 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.

### 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.

A single framebuffer-local dependency guarantees that only for a single framebuffer region, the first set of operations and availability operations happen-before visibility operations and the second set of operations. No ordering guarantees are made between framebuffer regions for a framebuffer-local dependency.

A framebuffer-global dependency guarantees that the first set of operations for all framebuffer regions happens-before the second set of operations for any framebuffer region.

*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 a single sample.

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_KHX` 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_KHX` 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 VkDeviceGroupRenderPassBeginInfoKHX::`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_KHX` 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_KHX` 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.

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 renderpass, 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 renderpass 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.

## 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.

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 an instance of the `VkFenceCreateInfo` structure which contains information about how the fence is to be created.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pFence` points 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 pointer to a valid `VkFenceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pFence` **must** be a 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` defines the initial state and behavior of the fence. Bits which **can** be set include:

  ```
  typedef enum VkFenceCreateFlagBits {
      VK_FENCE_CREATE_SIGNALED_BIT = 0x00000001,
  } VkFenceCreateFlagBits;
  ```

  If `flags` contains `VK_FENCE_CREATE_SIGNALED_BIT` then the fence object is created in the signaled state; otherwise it is created in the unsignaled state.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_FENCE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of VkFenceCreateFlagBits values

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 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 |
|---|---|
| `VK_SUCCESS` | The fence specified by `fence` is signaled. |
| `VK_NOT_READY` | The fence specified by `fence` is unsignaled. |
| `VK_DEVICE_LOST` | The device has been lost. See Lost Device. |

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_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.

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

- Any given element of `pFences` **must** not currently 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
- `pFences` **must** be a 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 previous queue submissions to the same queue via vkQueueSubmit.

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_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 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 an instance of the VkDeviceEventInfoEXT structure describing the event of interest to the application.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pFence` points 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 pointer to a valid `VkDeviceEventInfoEXT` structure
- `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pFence` **must** be a 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` specifies when the fence will be signaled. Possible values are:

```
typedef enum VkDeviceEventTypeEXT {
    VK_DEVICE_EVENT_TYPE_DISPLAY_HOTPLUG_EXT = 0,
} VkDeviceEventTypeEXT;
```

  - `VK_DEVICE_EVENT_TYPE_DISPLAY_HOTPLUG_EXT` occurs 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.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT`
- `pNext` **must** be `NULL`
- `deviceEvent` **must** be a valid VkDeviceEventTypeEXT value

To create a fence that will be signaled when an event occurs on a VkDisplayKHR object, call:

```
VkResult vkRegisterDisplayEventEXT(
    VkDevice                                    device,
    VkDisplayKHR                                display,
    const VkDisplayEventInfoEXT*                pDisplayEventInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFence*                                    pFence);
```

- `device` is a logical device associated with `display`
- `display` is the display on which the event **may** occur.
- `pDisplayEventInfo` is a pointer to an instance of the VkDisplayEventInfoEXT structure describing the event of interest to the application.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pFence` points 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
- `pDisplayEventInfo` **must** be a pointer to a valid `VkDisplayEventInfoEXT` structure
- `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pFence` **must** be a 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` specifies when the fence will be signaled. Possible values are:

  ```
  typedef enum VkDisplayEventTypeEXT {
      VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT = 0,
  } VkDisplayEventTypeEXT;
  ```

  - `VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT` occurs when the first pixel of the next display refresh cycle leaves the display engine for the display.

**Valid Usage (Implicit)**

- `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT`
- `pNext` **must** be `NULL`
- `displayEvent` **must** be a valid VkDisplayEventTypeEXT value

## 6.4. Semaphores

Semaphores are a synchronization primitive that **can** be used to insert a dependency between batches submitted to queues. Semaphores have two states - signaled and unsignaled. The state of a semaphore **can** be signaled after execution of a batch of commands is completed. A batch **can** wait for a semaphore to become signaled before it begins execution, and the semaphore is also unsignaled before the batch begins execution.

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 an instance of the `VkSemaphoreCreateInfo` structure which contains information about how the semaphore is to be created.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pSemaphore` points to a handle in which the resulting semaphore object is returned.

When created, the semaphore is in the unsignaled state.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a pointer to a valid `VkSemaphoreCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSemaphore` **must** be a 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`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

To create a semaphore object that can be exported to external handles, add the VkExportSemaphoreCreateInfoKHX structure to the `pNext` chain of the VkSemaphoreCreateInfo structure. The `VkExportSemaphoreCreateInfoKHX` structure is defined as:

```
typedef struct VkExportSemaphoreCreateInfoKHX {
    VkStructureType                          sType;
    const void*                              pNext;
    VkExternalSemaphoreHandleTypeFlagsKHX    handleTypes;
} VkExportSemaphoreCreateInfoKHX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of VkExternalSemaphoreHandleTypeFlagBitsKHX specifying one or more semaphore handle types the application **can** export from the resulting allocation. 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 VkExternalSemaphorePropertiesKHX.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHX`
- `pNext` **must** be `NULL`
- `handleTypes` **must** be a valid combination of VkExternalSemaphoreHandleTypeFlagBitsKHX values

To specify additional attributes of NT handles exported from a semaphore, add the VkExportSemaphoreWin32HandleInfoKHX structure to the `pNext` chain of the VkSemaphoreCreateInfo structure. The `VkExportSemaphoreWin32HandleInfoKHX` structure is defined as:

```
typedef struct VkExportSemaphoreWin32HandleInfoKHX {
    VkStructureType                 sType;
    const void*                     pNext;
    const SECURITY_ATTRIBUTES*      pAttributes;
    DWORD                           dwAccess;
    LPCWSTR                         name;
} VkExportSemaphoreWin32HandleInfoKHX;
```

- `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 UNICODE 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"[1]. 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_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX`

And

`GENERIC_ALL`

for handles of the following types:

`VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT_KHX`

**1**

https://msdn.microsoft.com/en-us/library/windows/desktop/ms686670.aspx

### Valid Usage

- If VkExportSemaphoreCreateInfoKHX::`handleTypes` does not include `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX` or VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT_KHX, VkExportSemaphoreWin32HandleInfoKHX **must** not be in the `pNext` chain of VkSemaphoreCreateInfo.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHX`
- `pNext` **must** be `NULL`
- If `pAttributes` is not `NULL`, `pAttributes` **must** be a pointer to a valid `SECURITY_ATTRIBUTES` value

To export a Windows handle representing the state of a semaphore, call:

```
VkResult vkGetSemaphoreWin32HandleKHX(
    VkDevice                                    device,
    VkSemaphore                                 semaphore,
    VkExternalSemaphoreHandleTypeFlagBitsKHX    handleType,
    HANDLE*                                     pHandle);
```

- `device` is the logical device that created `semaphore`.
- `semaphore` is the semaphore from which state will be exported.
- `handleType` is the type of handle requested.
- `pHandle` will return the Windows handle representing the semaphore state.

The properties of the handle returned depend on the value of `handleType`. See VkExternalSemaphoreHandleTypeFlagBitsKHX for a description of the properties of the defined external semaphore handle types.

For handle types defined as NT handles, the handles returned by `vkGetSemaphoreWin32HandleKHX` 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

- handleType **must** have been included in VkExportSemaphoreCreateInfoKHX::handleTypes when semaphore's current state was created.
- If handleType is defined as an NT handle, vkGetSemaphoreWin32HandleKHX **must** be called no more than once for each valid unique combination of semaphore and handleType.
- semaphore **must** not currently have its state replaced by imported semaphore state as described below in Importing Semaphore State unless that imported semaphore state's handle type was included in VkExternalSemaphorePropertiesKHX::exportFromImportedHandleTypes.
- If handleType refers to a handle type with temporary import semantics, as defined below in Importing Semaphore State, there **must** be no queue waiting on semaphore.
- If handleType refers to a handle type with temporary import 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)

- device **must** be a valid VkDevice handle
- semaphore **must** be a valid VkSemaphore handle
- handleType **must** be a valid VkExternalSemaphoreHandleTypeFlagBitsKHX value
- pHandle **must** be a pointer to a HANDLE value
- semaphore **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

To export a POSIX file descriptor representing the state of a semaphore, call:

```
VkResult vkGetSemaphoreFdKHX(
    VkDevice                                    device,
    VkSemaphore                                 semaphore,
    VkExternalSemaphoreHandleTypeFlagBitsKHX    handleType,
    int*                                        pFd);
```

- `device` is the logical device that created `semaphore`.
- `semaphore` is the semaphore from which state will be exported.
- `handleType` is the type of handle requested.
- `pFd` will return the file descriptor representing the semaphore state.

The properties of the file descriptor returned depend on the value of `handleType`. See VkExternalSemaphoreHandleTypeFlagBitsKHX for a description of the properties of the defined external semaphore handle types.

Each call to `vkGetSemaphoreFdKHX` **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 Vulkan semaphore state from it.

### Valid Usage

- `handleType` **must** have been included in VkExportSemaphoreCreateInfoKHX::`handleTypes` when `semaphore`'s current state was created.
- `semaphore` **must** not currently have its state replaced by imported semaphore state as described below in Importing Semaphore State unless that imported semaphore state's handle type was included in VkExternalSemaphorePropertiesKHX ::`exportFromImportedHandleTypes`.
- If `handleType` refers to a handle type with temporary import semantics, as defined below in Importing Semaphore State, there **must** be no queue waiting on `semaphore`.
- If `handleType` refers to a handle type with temporary import semantics, `semaphore` **must** be signaled, or have an associated semaphore signal operation pending execution.
- `handleType` **must** be defined as a POSIX file descriptor handle.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `semaphore` **must** be a valid `VkSemaphore` handle
- `handleType` **must** be a valid VkExternalSemaphoreHandleTypeFlagBitsKHX value
- `pFd` **must** be a pointer to a `int` value
- `semaphore` **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

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 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.

The first synchronization scope includes every command submitted in the same batch. Semaphore signal operations that are defined by vkQueueSubmit additionally include all batches previously submitted to the same queue via vkQueueSubmit, including batches that are submitted in the same queue submission command, but at a lower index within the array of batches.

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 & Unsignaling

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 unsignal operations* which set the semaphores to the unsignaled state.

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 batches subsequently submitted to the same queue via vkQueueSubmit, including batches that are submitted in the same queue submission command, but at a higher index within the array of batches.

The first access scope is empty.

The second access scope includes all memory access performed by the device.

The semaphore unsignal 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 fences or events, the act of waiting for a semaphore also unsignals that semaphore. If two operations are separately specified to wait for the same semaphore, and there are no other execution dependencies between those operations, behaviour is undefined. An execution dependency **must** be present that guarantees that the semaphore unsignal operation for the first of those waits, happens-before the semaphore is signalled again, and before the second unsignal operation. 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 and unsignal operation is submitted to a queue:

- The semaphore **must** be signaled, or have an associated semaphore signal operation that is pending execution.
- There **must** be no other queue waiting on the same semaphore when the operation executes.

Generally, the implementation behavior will be undefined when applications fail to ensure these requirements are met. However, to avoid requiring Vulkan instances that import semaphore state to trust the source of that state, the side effects of violating the requirements are better defined in certain cases. When the semaphore that violates the requirements is using imported state at the time of the violation, the implementation **must** ensure side effects are limited to one or more of the following:

- Returning the error code `VK_ERROR_INITIALIZATION_FAILED` from the command which resulted in the violation.
- Allowing the logical device on which the violation occured to enter an undefined state immediately or in the future and returning the error code VK_ERROR_DEVICE_LOST from any subsequent command, including the one causing the violation, after the device enters such a state.
- 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 semaphore becomes undefined. The semaphore **must** be destroyed or have its state replaced by an import operation to again meet valid usage requirements.

### 6.4.4. Importing Semaphore State

Applications **may** import semaphore state into an existing semaphore using an external semaphore handle. Doing so releases any existing state as if vkDestroySemaphore had been called on the target semaphore. As such, similar usage restrictions to those applied to `vkDestroySemaphore` are applied to any command that imports semaphore state. The effects of the import operation will be either temporary or permanent, depending on the type of handle used to perform the import. If the import is temporary, the implementation **must** restore the semaphore to its prior permanent state when executing the next semaphore signal operation. If the import is permanent, the signaled status of all semaphores sharing the same semaphore state **must** be equivalent. Semaphore signaling and waiting operations performed on any semaphore in the set of semaphores sharing semaphore state **must** behave the same as if the set were a single semaphore. Passing a semaphore to vkAcquireNextImageKHR is equivalent to temporarily importing semaphore state to that semaphore. When a semaphore is using imported state, its VkExportSemaphoreCreateInfoKHX ::`handleTypes` value is that specified when creating the semaphore from which the state was exported, rather than that specified when creating the semaphore.

> *Note*
>
> Because the exportable handle types of an imported semaphore correspond to its current imported state, 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 temporarily imported state from vkAcquireNextImageKHR.

When importing semaphore state, 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 valid semaphore state 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 state import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX.

To import semaphore state from a Windows handle, call:

```
VkResult vkImportSemaphoreWin32HandleKHX(
    VkDevice                                    device,
    const VkImportSemaphoreWin32HandleInfoKHX*  pImportSemaphoreWin32HandleInfo);
```

- `device` is the logical device that created the semaphore.
- `pImportSemaphoreWin32HandleInfo` points to a VkImportSemaphoreWin32HandleInfoKHX structure specifying the semaphore and import parameters.

Importing semaphore state 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.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pImportSemaphoreWin32HandleInfo` **must** be a pointer to a valid `VkImportSemaphoreWin32HandleInfoKHX` structure

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX`

The `VkImportSemaphoreWin32HandleInfoKHX` structure is defined as:

```
typedef struct VkImportSemaphoreWin32HandleInfoKHX {
    VkStructureType                           sType;
    const void*                               pNext;
    VkSemaphore                               semaphore;
    VkExternalSemaphoreHandleTypeFlagsKHX     handleType;
    HANDLE                                    handle;
} VkImportSemaphoreWin32HandleInfoKHX;
```

- `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 state will be imported.
- `handleType` specifies the type of `handle`.
- `handle` is the external handle to import.

The handle types supported by `handleType` are:

*Table 6. Handle Type Permanence for VkImportSemaphoreWin32HandleInfoKHX*

| Handle Type | Permanence |
|---|---|
| VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX | Permanent |
| VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHX | Permanent |
| VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT_KHX | Permanent |

### Valid Usage

- `handleType` **must** be a value included in the Handle Type Permanence for VkImportSemaphoreWin32HandleInfoKHX table.
- The semaphore from which `handleType` was exported **must** have been created on the same underlying physical device as `semaphore`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHX`
- `pNext` **must** be `NULL`
- `semaphore` **must** be a valid `VkSemaphore` handle
- `handleType` **must** be a valid combination of VkExternalSemaphoreHandleTypeFlagBitsKHX values
- `handleType` **must** not be `0`

To import semaphore state from a POSIX file descriptor, call:

```
VkResult vkImportSemaphoreFdKHX(
    VkDevice                                    device,
    const VkImportSemaphoreFdInfoKHX*           pImportSemaphoreFdInfo);
```

- `device` is the logical device that created the semaphore.

- pImportSemaphoreFdInfo points to a VkImportSemaphoreFdInfoKHX structure specifying the semaphore and import parameters.

Importing semaphore state 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.

### 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 pointer to a valid VkImportSemaphoreFdInfoKHX structure

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX

The VkImportSemaphoreFdInfoKHX structure is defined as:

```
typedef struct VkImportSemaphoreFdInfoKHX {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkSemaphore                                 semaphore;
    VkExternalSemaphoreHandleTypeFlagBitsKHX    handleType;
    int                                         fd;
} VkImportSemaphoreFdInfoKHX;
```

- 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 state will be imported.
- handleType specifies the type of fd.
- fd is the external handle to import.

The handle types supported by handleType are:

*Table 7. Handle Type Permanence for VkImportSemaphoreFdInfoKHX*

| Handle Type | Permanence |
|---|---|
| VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT_KHX | Permanent |
| VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_FENCE_FD_BIT_KHX | Temporary |

### Valid Usage

- `handleType` **must** be a value included in the Handle Type Permanence for VkImportSemaphoreFdInfoKHX table.
- The semaphore from which `handleType` was exported **must** have been created on the same underlying physical device as `semaphore`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHX`
- `pNext` **must** be `NULL`
- `semaphore` **must** be a valid `VkSemaphore` handle
- `handleType` **must** be a valid VkExternalSemaphoreHandleTypeFlagBitsKHX value

## 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 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 an instance of the `VkEventCreateInfo` structure which contains information about how the event is to be created.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pEvent` points 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 pointer to a valid `VkEventCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pEvent` **must** be a 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`

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 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 8. Event Object Status Codes*

| Status | Meaning |
|---|---|
| `VK_EVENT_SET` | The event specified by `event` is signaled. |
| `VK_EVENT_RESET` | The event specified by `event` 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                             commandBuffer,
    VkEvent                                     event,
    VkPipelineStageFlags                        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 every command previously submitted to the same queue, including those in the same command buffer and batch. 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.

### 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<br>Secondary | Outside | Graphics<br>compute | |

To set the state of an event to unsignaled from a device, call:

```
void vkCmdResetEvent(
    VkCommandBuffer                             commandBuffer,
    VkEvent                                     event,
    VkPipelineStageFlags                        stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be unsignaled.
- `stageMask` specifies 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 every command previously submitted to the same queue, including those in the same command buffer and batch. 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.

## 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<br>Secondary | Outside | Graphics<br>compute | |

To wait for one or more events to enter the signaled state on a device, call:

```
void vkCmdWaitEvents(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    eventCount,
    const VkEvent*                              pEvents,
    VkPipelineStageFlags                        srcStageMask,
    VkPipelineStageFlags                        dstStageMask,
    uint32_t                                    memoryBarrierCount,
    const VkMemoryBarrier*                      pMemoryBarriers,
    uint32_t                                    bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier*                pBufferMemoryBarriers,
    uint32_t                                    imageMemoryBarrierCount,
    const VkImageMemoryBarrier*                 pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `eventCount` is the length of the `pEvents` array.
- `pEvents` is an array of event object handles to wait on.
- `srcStageMask` is the source stage mask
- `dstStageMask` is 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, and subsequent commands.

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 were previously submitted to the same queue 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 commands subsequently submitted to the same queue, including those in the same command buffer and batch. 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.

*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.
- Any given 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.
- Any given 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.
- commandBuffer's current device mask **must** include exactly one physical device.

### Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pEvents` **must** be a 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 pointer to an array of `memoryBarrierCount` valid `VkMemoryBarrier` structures
- If `bufferMemoryBarrierCount` is not `0`, `pBufferMemoryBarriers` **must** be a pointer to an array of `bufferMemoryBarrierCount` valid `VkBufferMemoryBarrier` structures
- If `imageMemoryBarrierCount` is not `0`, `pImageMemoryBarriers` **must** be a 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<br>Secondary | Both | Graphics<br>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                             commandBuffer,
    VkPipelineStageFlags                        srcStageMask,
    VkPipelineStageFlags                        dstStageMask,
    VkDependencyFlags                           dependencyFlags,
    uint32_t                                    memoryBarrierCount,
    const VkMemoryBarrier*                      pMemoryBarriers,
    uint32_t                                    bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier*                pBufferMemoryBarriers,
    uint32_t                                    imageMemoryBarrierCount,
    const VkImageMemoryBarrier*                 pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `srcStageMask` defines a source stage mask.
- `dstStageMask` defines a destination stage mask.
- `dependencyFlags` is a bitmask of VkDependencyFlagBits. The bits that **can** be included in `dependencyFlags` are:

  ```
  typedef enum VkDependencyFlagBits {
      VK_DEPENDENCY_BY_REGION_BIT = 0x00000001,
      VK_DEPENDENCY_VIEW_LOCAL_BIT_KHX = 0x00000002,
      VK_DEPENDENCY_DEVICE_GROUP_BIT_KHX = 0x00000004,
  } VkDependencyFlagBits;
  ```

  - `VK_DEPENDENCY_BY_REGION_BIT` signifies that dependencies will be framebuffer-local.
- `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 every command submitted to the same queue before it, including those in the same command buffer and batch. If vkCmdPipelineBarrier was recorded inside a render pass instance, the first synchronization scope includes only commands submitted before it 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 every command submitted to the same queue after it, including those in the same command buffer and batch. If vkCmdPipelineBarrier was recorded inside a render pass instance, the second synchronization scope includes only commands submitted after it 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 a VkSubpassDependency instance in pDependencies that expresses a dependency from the current subpass to itself.
- If vkCmdPipelineBarrier is called within a render pass instance, srcStageMask **must** contain a subset of the bit values in the srcStageMask member of that instance of VkSubpassDependency
- If vkCmdPipelineBarrier is called within a render pass instance, dstStageMask **must** contain a subset of the bit values in the dstStageMask member of that instance of VkSubpassDependency
- If vkCmdPipelineBarrier is called within a render pass instance, the srcAccessMask of any element of pMemoryBarriers or pImageMemoryBarriers **must** contain a subset of the bit values the srcAccessMask member of that instance of VkSubpassDependency
- If vkCmdPipelineBarrier is called within a render pass instance, the dstAccessMask of any element of pMemoryBarriers or pImageMemoryBarriers **must** contain a subset of the bit values the dstAccessMask member of that instance of VkSubpassDependency
- If vkCmdPipelineBarrier is called within a render pass instance, dependencyFlags **must** be equal to the dependencyFlags member of that instance of VkSubpassDependency
- 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 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 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.
- Any given 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.
- Any given 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.
- If vkCmdPipelineBarrier is called outside of a render pass instance, dependencyFlags **must** not include VK_DEPENDENCY_VIEW_LOCAL_BIT_KHX

## 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 pointer to an array of memoryBarrierCount valid VkMemoryBarrier structures
- If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers **must** be a pointer to an array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier structures
- If imageMemoryBarrierCount is not 0, pImageMemoryBarriers **must** be a 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<br>Secondary | Both | Transfer<br>graphics<br>compute | |

### 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. Self-dependencies **must** not have any earlier pipeline stages depend on any later pipeline stages (according to the order of graphics pipeline stages), unless all of the stages are framebuffer-space stages. 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_KHX`.

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_KHX` 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` defines a source access mask.
- `dstAccessMask` defines 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 an 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` defines a source access mask.
- `dstAccessMask` defines 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 `VK_QUEUE_FAMILY_EXTERNAL_KHX`
- 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 VK_QUEUE_FAMILY_EXTERNAL_KHX (see Queue Family Properties)
- 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 VK_QUEUE_FAMILY_EXTERNAL_KHX (see Queue Family Properties)
- If `buffer` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are valid queue families, at least one of them **must** be the same as the family of the queue that will execute this barrier

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER`
- `pNext` **must** be `NULL`
- `srcAccessMask` **must** be a valid combination of VkAccessFlagBits values
- `dstAccessMask` **must** be a valid combination of VkAccessFlagBits values
- `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` defines a source access mask.
- `dstAccessMask` defines 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.

## 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 `VK_QUEUE_FAMILY_EXTERNAL_KHX`
- 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 VK_QUEUE_FAMILY_EXTERNAL_KHX (see Queue Family Properties).
- 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 VK_QUEUE_FAMILY_EXTERNAL_KHX (see Queue Family Properties).
- If `image` was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are valid queue families, at least one of them **must** be the same as the family of the queue that will execute this barrier
- `subresourceRange` **must** be a valid image subresource range for the image (see Image Views)
- If `image` has a depth/stencil format with both depth and stencil components, then `aspectMask` member of `subresourceRange` **must** include both `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_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_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

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER`
- `pNext` **must** be `NULL`
- `srcAccessMask` **must** be a valid combination of VkAccessFlagBits values
- `dstAccessMask` **must** be a valid combination of VkAccessFlagBits values
- `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_KHX` represents any queue external to the resource's current Vulkan instance. 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.

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), 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. `dstStageMask` 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.

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), on a queue from the destination 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. `srcStageMask` 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-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 an `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.

## 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

### 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 every command submitted in the same queue submission command, and all future submissions to the same queue.

The first access scope includes all host writes to mappable device memory that are either coherent, or have been flushed with vkFlushMappedMemoryRanges.

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 VkDeviceGroupSubmitInfoKHX and VkDeviceGroupBindSparseInfoKHX 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).

# 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, or input attachment for that subpass (as determined by the `pColorAttachments`, `pDepthStencilAttachment`, `pResolveAttachments`, 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 an instance of the VkRenderPassCreateInfo structure that describes the parameters of the render pass.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pRenderPass` points 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 pointer to a valid `VkRenderPassCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pRenderPass` **must** be a 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, or zero indicating no attachments. Attachments are referred to by zero-based indices in the range [0,attachmentCount).
- pAttachments points to an array of attachmentCount number of VkAttachmentDescription structures describing properties of the attachments, or NULL if attachmentCount is zero.
- subpassCount is the number of subpasses to create for this render pass. Subpasses are referred to by zero-based indices in the range [0,subpassCount). A render pass **must** have at least one subpass.
- pSubpasses points to an array of subpassCount number of VkSubpassDescription structures describing properties of the subpasses.
- dependencyCount is the number of dependencies between pairs of subpasses, or zero indicating no dependencies.
- pDependencies points to an array of dependencyCount number of VkSubpassDependency structures describing dependencies between pairs of subpasses, or NULL if dependencyCount is zero.

## 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 `VkAttachmentDescription` 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`
- The value of any element of the `pPreserveAttachments` member in any given element of `pSubpasses` **must** not be `VK_ATTACHMENT_UNUSED`
- 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 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 source subpass.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- If `attachmentCount` is not `0`, `pAttachments` **must** be a pointer to an array of `attachmentCount` valid `VkAttachmentDescription` structures
- `pSubpasses` **must** be a pointer to an array of `subpassCount` valid `VkSubpassDescription` structures
- If `dependencyCount` is not `0`, `pDependencies` **must** be a pointer to an array of `dependencyCount` valid `VkSubpassDependency` structures
- `subpassCount` **must** be greater than `0`

If the `VkRenderPassCreateInfo::pNext` list includes a `VkRenderPassMultiviewCreateInfoKHX` structure, then that structure includes an array of view masks, view offsets, and correlation masks for the render pass.

The `VkRenderPassMultiviewCreateInfoKHX` structure is defined as:

```
typedef struct VkRenderPassMultiviewCreateInfoKHX {
    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;
} VkRenderPassMultiviewCreateInfoKHX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `subpassCount` is zero or is the number of subpasses in the render pass.
- `pViewMasks` points to an array of `subpassCount` number of 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` points 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 a number of correlation masks.
- `pCorrelationMasks` is an array of 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_KHX` 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

- If `subpassCount` is not zero, `subpassCount` **must** be equal to the `subpassCount` in the `VkRenderPassCreateInfo` structure at the start of the chain
- If `dependencyCount` is not zero, `dependencyCount` **must** be equal to the `dependencyCount` in the `VkRenderPassCreateInfo` structure at the start of the chain
- Each view index **must** not be set in more than one element of `pCorrelationMasks`
- If an element of `pViewOffsets` is non-zero, the corresponding VkSubpassDependency structure **must** have different values of `srcSubpass` and `dstSubpass`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO_KHX`
- `pNext` **must** be `NULL`
- If `subpassCount` is not `0`, `pViewMasks` **must** be a pointer to an array of `subpassCount` `uint32_t` values
- If `dependencyCount` is not `0`, `pViewOffsets` **must** be a pointer to an array of `dependencyCount` `int32_t` values
- If `correlationMaskCount` is not `0`, `pCorrelationMasks` **must** be a pointer to an array of `correlationMaskCount` `uint32_t` values

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 describing additional properties of the attachment. Bits which **can** be set include:

```
typedef enum VkAttachmentDescriptionFlagBits {
    VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT = 0x00000001,
} VkAttachmentDescriptionFlagBits;
```

- `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` specifies how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used:

```
typedef enum VkAttachmentLoadOp {
    VK_ATTACHMENT_LOAD_OP_LOAD = 0,
    VK_ATTACHMENT_LOAD_OP_CLEAR = 1,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE = 2,
} VkAttachmentLoadOp;
```

  - `VK_ATTACHMENT_LOAD_OP_LOAD` means 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` means 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` means 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.

- storeOp specifies how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used:

```
typedef enum VkAttachmentStoreOp {
    VK_ATTACHMENT_STORE_OP_STORE = 0,
    VK_ATTACHMENT_STORE_OP_DONT_CARE = 1,
} VkAttachmentStoreOp;
```

  - VK_ATTACHMENT_STORE_OP_STORE means 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 means 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.

- stencilLoadOp specifies how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used, and **must** be one of the same values allowed for loadOp above.
- stencilStoreOp specifies how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used, and **must** be one of the same values allowed for storeOp above.
- 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. During a render pass instance, an attachment **can** use a different layout in each subpass, if desired.

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 value in an attachment used by a subpass happens-before any command recorded into that subpass reads from that value. 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.

Store operations for each value in an attachment used by a subpass happen-after any command recorded into that subpass writes to that value. 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

### 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

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 indicating usage of the subpass. Bits which **can** be set include:

  ```
  typedef enum VkSubpassDescriptionFlagBits {
      VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX = 0x00000001,
      VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX = 0x00000002,
  } VkSubpassDescriptionFlagBits;
  ```

  - VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX indicates 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 indicates 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.

- pipelineBindPoint is a VkPipelineBindPoint value specifying whether this is a compute or graphics subpass. Currently, only graphics subpasses are supported.
- inputAttachmentCount is the number of input attachments.
- pInputAttachments is an array of VkAttachmentReference structures (defined below) that lists which of the render pass's attachments **can** be read in the shader during the subpass, and what layout each attachment will be in during the subpass. Each element of the array corresponds to

an input attachment unit number in the shader, i.e. if the shader declares an input variable `layout(input_attachment_index=X, set=Y, binding=Z)` then it uses the attachment provided in `pInputAttachments`[X]. Input attachments **must** also be bound to the pipeline with a descriptor set, with the input attachment descriptor written in the location (set=Y, binding=Z).

- `colorAttachmentCount` is the number of color attachments.
- `pColorAttachments` is an array of `colorAttachmentCount` VkAttachmentReference structures that lists which of the render pass's attachments will be used as color attachments in the subpass, and what layout each attachment will be in during the subpass. Each element of the array corresponds to a fragment shader output location, i.e. if the shader declared an output variable `layout(location=X)` then it uses the attachment provided in `pColorAttachments`[X].
- `pResolveAttachments` is `NULL` or an array of `colorAttachmentCount` VkAttachmentReference structures that lists which of the render pass's attachments are resolved to at the end of the subpass, and what layout each attachment will be in during the multisample resolve operation. 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 to the same pixel location in the corresponding resolve attachments, unless the resolve attachment index is `VK_ATTACHMENT_UNUSED`. If the first use of an attachment in a render pass is as a resolve attachment, then the `loadOp` is effectively ignored as the resolve is guaranteed to overwrite all pixels in the render area.
- `pDepthStencilAttachment` is a pointer to a VkAttachmentReference specifying which attachment will be used for depth/stencil data and the layout it will be in during the subpass. Setting the attachment index to `VK_ATTACHMENT_UNUSED` or leaving this pointer as `NULL` indicates that no depth/stencil attachment will be used in the subpass.
- `preserveAttachmentCount` is the number of preserved attachments.
- `pPreserveAttachments` is an array of `preserveAttachmentCount` render pass attachment indices describing the attachments that are not used by a subpass, but whose contents **must** be preserved throughout 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 $\mathbf{S_1}$ that uses or preserves the attachment, and a subpass dependency from $\mathbf{S_1}$ 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 $\mathbf{S_1}$ 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 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`, the sample count of each element of `pColorAttachments` **must** be anything other than `VK_SAMPLE_COUNT_1_BIT`
- Any given element of `pResolveAttachments` **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` and `pDepthStencilAttachment` that are not `VK_ATTACHMENT_UNUSED` **must** have the same sample count
- If any input attachments are `VK_ATTACHMENT_UNUSED`, then any pipelines bound during the subpass **must** not access those input attachments from the fragment shader
- 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 as both an input attachment and a color or depth/stencil attachment, 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 pointer to an array of `inputAttachmentCount` valid `VkAttachmentReference` structures
- If `colorAttachmentCount` is not `0`, `pColorAttachments` **must** be a pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures
- If `colorAttachmentCount` is not `0`, and `pResolveAttachments` is not `NULL`, `pResolveAttachments` **must** be a pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures
- If `pDepthStencilAttachment` is not `NULL`, `pDepthStencilAttachment` **must** be a pointer to a valid `VkAttachmentReference` structure
- If `preserveAttachmentCount` is not `0`, `pPreserveAttachments` **must** be a pointer to an array of `preserveAttachmentCount` `uint32_t` values

The `VkAttachmentReference` structure is defined as:

```
typedef struct VkAttachmentReference {
    uint32_t         attachment;
    VkImageLayout    layout;
} VkAttachmentReference;
```

- `attachment` is the index of the attachment of the render pass, and corresponds to the index of the corresponding element in the `pAttachments` array of the `VkRenderPassCreateInfo` structure. If any color or depth/stencil attachments are `VK_ATTACHMENT_UNUSED`, then no writes occur for those attachments.
- `layout` is a VkImageLayout value specifying the layout the attachment uses during the subpass.

### Valid Usage

- `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` defines a source stage mask.
- `dstStageMask` defines a destination stage mask.
- `srcAccessMask` defines a source access mask.
- `dstAccessMask` defines 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 submitted to the queue before the render pass instance began. 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 submitted after the render pass instance is ended. 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.

## Valid Usage

- If srcSubpass is not VK_SUBPASS_EXTERNAL, srcStageMask **must** not include VK_PIPELINE_STAGE_HOST_BIT
- If dstSubpass is not VK_SUBPASS_EXTERNAL, dstStageMask **must** not include VK_PIPELINE_STAGE_HOST_BIT
- 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, srcStageMask and dstStageMask **must** only contain one of 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_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, or VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- 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_KHX, then both srcSubpass and dstSubpass **must** not equal VK_SUBPASS_EXTERNAL
- If dependencyFlags includes VK_DEPENDENCY_VIEW_LOCAL_BIT_KHX, then the render pass **must** have multiview enabled
- 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_KHX

### 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 × 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 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 renderpass, 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 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.

If two subpasses use the same attachment in different layouts, and both layouts are read-only, 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_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.

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 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

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` points to a VkFramebufferCreateInfo structure which describes additional information about framebuffer creation.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pFramebuffer` points to a `VkFramebuffer` handle in which the resulting framebuffer object is returned.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a pointer to a valid `VkFramebufferCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pFramebuffer` **must** be a 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 reserved for future use.
- `renderPass` is a render pass that defines what render passes the framebuffer will be compatible with. See Render Pass Compatibility for details.
- `attachmentCount` is the number of attachments.
- `pAttachments` is an array of `VkImageView` handles, each of which will be used as the corresponding attachment in a render pass instance.
- `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.

Image subresources used as attachments **must** not be used via any non-attachment usage for the duration of a render pass instance.

> *Note*
>
> This restriction means 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`
- Any given 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`
- Any given 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`
- Any given 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`
- Any given element of `pAttachments` **must** have been created with an VkFormat value that matches the VkFormat specified by the corresponding `VkAttachmentDescription` in `renderPass`
- Any given element of `pAttachments` **must** have been created with a `samples` value that matches the `samples` value specified by the corresponding `VkAttachmentDescription` in `renderPass`
- Any given element of `pAttachments` **must** have dimensions at least as large as the corresponding framebuffer dimension
- Any given element of `pAttachments` **must** only specify a single mip level
- Any given 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`
- Any given element of `pAttachments` that is a 2D or 2D array image view taken from a 3D image **must** not be a depth/stencil format

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `renderPass` **must** be a valid `VkRenderPass` handle
- If `attachmentCount` is not `0`, `pAttachments` **must** be a pointer to an array of `attachmentCount` valid `VkImageView` handles
- Both of `renderPass`, and the elements of `pAttachments` that are valid handles **must** have been created, allocated, or retrieved from the same `VkDevice`

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 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                             commandBuffer,
    const VkRenderPassBeginInfo*                pRenderPassBegin,
    VkSubpassContents                           contents);
```

- `commandBuffer` is the command buffer in which to record the command.
- `pRenderPassBegin` is a pointer to a VkRenderPassBeginInfo structure (defined below) which indicates the render pass to begin an instance of, and the framebuffer the instance uses.
- `contents` specifies how the commands in the first subpass will be provided, and is one of the values:

  ```
  typedef enum VkSubpassContents {
      VK_SUBPASS_CONTENTS_INLINE = 0,
      VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS = 1,
  } VkSubpassContents;
  ```

  If `contents` is `VK_SUBPASS_CONTENTS_INLINE`, 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. If `contents` is `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS`, 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`.

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 subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` set
- 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_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` then the corresponding attachment image subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` set
- 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 subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` set
- 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 subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` set
- 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 subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` set
- 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.

## Valid Usage (Implicit)

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

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 an array of 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, otherwise the pixels outside of the render area become undefined and shader side effects **may** occur for fragments outside 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`
- If `clearValueCount` is not `0`, `pClearValues` **must** be a pointer to an array of `clearValueCount` valid `VkClearValue` unions
- `renderPass` **must** be compatible with the `renderPass` member of the `VkFramebufferCreateInfo` structure specified when creating `framebuffer`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkDeviceGroupRenderPassBeginInfoKHX
- `renderPass` **must** be a valid `VkRenderPass` handle
- `framebuffer` **must** be a valid `VkFramebuffer` handle
- Both of `framebuffer`, and `renderPass` **must** have been created, allocated, or retrieved from the same `VkDevice`

If the pNext list of VkRenderPassBeginInfo includes a VkDeviceGroupRenderPassBeginInfoKHX structure, then that structure includes a device mask and set of render areas for the render pass instance.

The VkDeviceGroupRenderPassBeginInfoKHX structure is defined as:

```
typedef struct VkDeviceGroupRenderPassBeginInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           deviceMask;
    uint32_t           deviceRenderAreaCount;
    const VkRect2D*    pDeviceRenderAreas;
} VkDeviceGroupRenderPassBeginInfoKHX;
```

- 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 an array of structures of type VkRect2D 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. Render pass attachment load, store, and resolve operations only apply to physical devices included in the device mask. Subpass dependencies only apply to the physical devices in the device mask.

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 VkDeviceGroupCommandBufferBeginInfoKHX::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_KHX`
- If `deviceRenderAreaCount` is not `0`, `pDeviceRenderAreas` **must** be a pointer to an array of `deviceRenderAreaCount` `VkRect2D` structures

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` points 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`+`offset.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`+`offset.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. 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 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. 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

### 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 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

### 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 |

# 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` parameter is a pointer to an instance of the `VkShaderModuleCreateInfo` structure.

- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pShaderModule points 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 pointer to a valid VkShaderModuleCreateInfo structure
- If pAllocator is not NULL, pAllocator **must** be a pointer to a valid VkAllocationCallbacks structure
- pShaderModule **must** be a 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 points 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 points 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_glsl extension specification
- If pCode points 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 points to GLSL code, it **must** be valid GLSL code written to the GL_KHR_vulkan_glsl 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 that are listed as not required by the implementation, the relevant feature **must** be enabled, as listed in the SPIR-V Environment appendix

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO
- pNext **must** be NULL
- flags **must** be 0
- pCode **must** be a pointer to an array of $codeSize \frac{e}{4}$ uint32_t values

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 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 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 order of invocations of the same shader type are 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 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.

Stores issued to different memory locations within a single shader invocation **may** not be visible to other invocations, or **may** not become visible in the order they were performed.

The `OpMemoryBarrier` instruction **can** be used to provide stronger ordering of reads and writes performed by a single invocation. `OpMemoryBarrier` guarantees that any memory transactions issued by the shader invocation prior to the instruction complete prior to the memory transactions issued after the instruction. Memory barriers are needed for algorithms that require multiple invocations to access the same memory and require the operations to be performed in a partially-defined relative order. For example, if one shader invocation does a series of writes, followed by an `OpMemoryBarrier` instruction, followed by another write, then the results of the series of writes before the barrier become visible to other shader invocations at a time earlier or equal to when the results of the final write become visible to those invocations. In practice it means that another invocation that sees the results of the final write would also see the previous writes. Without the memory barrier, the final write **may** be visible before the previous writes.

Writes that are the result of shader stores through a variable decorated with `Coherent` automatically have available writes to the same buffer, buffer view, or image view made visible to them, and are themselves automatically made available to access by the same buffer, buffer view, or image view. Reads that are the result of shader loads through a variable decorated with `Coherent` automatically have available writes to the same buffer, buffer view, or image view made visible to them. The order that coherent writes to different locations become available is undefined, unless enforced by a memory barrier instruction or other memory dependency.

> *Note*
>
> Explicit memory dependencies **must** still be used to guarantee availability and visibility for access via other buffers, buffer views, or image views.

The built-in atomic memory transaction instructions **can** be used to read and write a given memory address atomically. While built-in atomic functions issued by multiple shader invocations are

executed in undefined order relative to each other, these functions perform both a read and a write of a memory address and guarantee that no other memory transaction will write to the underlying memory between the read and write. Atomic operations ensure automatic availability and visibility for writes and reads in the same way as those to `Coherent` variables.

*Example 1. Note*

Memory accesses performed on different resource descriptors with the same memory backing **may** not be well-defined even with the `Coherent` decoration or via atomics, due to things such as image layouts or ownership of the resource - as described in the Synchronization and Cache Control chapter.

*Note*

Atomics allow shaders to use shared global addresses for mutual exclusion or as counters, among other uses.

# 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. 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 **must** include a vertex shader, and the vertex shader stage is always the first shader stage in the graphics pipeline.

## 8.5.1. 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`).

# 8.6. 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`.

## 8.6.1. 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.

# 8.7. 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.

## 8.7.1. 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.

# 8.8. 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.

## 8.8.1. 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. The number of geometry shader invocations per input primitive is determined from the invocation count of the geometry shader specified by the `OpExecutionMode Invocations` in the geometry shader. If the invocation count is not specified, then a default of one invocation is executed. If the subpass includes multiple views in its view mask, the shader **may** be invoked separately for each view.

# 8.9. 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.

## 8.9.1. 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.

The number of fragment shader invocations produced per-pixel is determined as follows:

- If per-sample shading is enabled, the fragment shader is invoked once per covered sample.
- Otherwise, the fragment shader is invoked at least once per fragment but no more than once per covered sample.

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.

### 8.9.2. 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.

## 8.10. 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.

## 8.11. 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 pixel and a single value **may** be assigned to each

sample within the pixel.

`Centroid` and `Sample` **can** be used to control the location and frequency of the sampling of the decorated fragment shader input. If a fragment shader input is decorated with `Centroid`, a single value **may** be assigned to that variable for all samples in the pixel, but that value **must** be interpolated to a location that lies in both the pixel and in the primitive being rendered, including any of the pixel's samples covered by the primitive. Because the location at which the variable is interpolated **may** be different in neighboring pixels, and derivatives **may** be computed by computing differences between neighboring pixels, derivatives of centroid-sampled inputs **may** be less accurate than those for non-centroid interpolated variables. If a fragment shader input is decorated with `Sample`, a separate value **must** be assigned to that variable for each covered sample in the pixel, and that value **must** be sampled at the location of the individual sample. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the pixel 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.

## 8.12. 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 that contains 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.

## 8.13. 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) for a fragment shader is the set of invocations generated by a single primitive (point, line, or triangle), including any helper invocations generated by that primitive. Derivatives are undefined for a sampled image instruction if the instruction is in flow control that is not uniform across the derivative group.

# 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* or a *compute pipeline*.

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.

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 1. 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 device state in command buffers. Any pipeline object state that is marked as dynamic is not applied to the device state when the pipeline is bound. Dynamic state not set by binding the pipeline object **can** be modified at any time and persists for the lifetime of the command buffer, or until modified by another dynamic state command or another pipeline bind. No state, including dynamic state, is inherited from one command buffer to another. Only dynamic state that is **required** for the operations performed in the command buffer needs to be set. For example, if blending is disabled by the pipeline state then the dynamic color blend constants do not need to be specified in the command buffer, even if this state is marked as dynamic in the pipeline state object. If a new pipeline object is bound with state not marked as dynamic after a previous

pipeline object with that same state as dynamic, the new pipeline object state will override the dynamic state. Modifying dynamic state that is not set as dynamic by the pipeline state object will lead to undefined results.

Compute and graphics 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 an array of `VkComputePipelineCreateInfo` structures.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pPipelines` is a pointer to an array in which the resulting compute pipeline objects are returned.

## Valid Usage

- If the `flags` member of any given 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 given 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 pointer to an array of `createInfoCount` valid `VkComputePipelineCreateInfo` structures
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pPipelines` **must** be a 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` provides options for pipeline creation, and is of type VkPipelineCreateFlagBits.
- `stage` is a VkPipelineShaderStageCreateInfo 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.

`stage` points to a structure of type `VkPipelineShaderStageCreateInfo`.

### 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`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `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 **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 reserved for future use.
- `stage` names a single pipeline stage. Bits which **can** be set include:

  ```
  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,
  } VkShaderStageFlagBits;
  ```

- `module` is a `VkShaderModule` object that contains 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 VkSpecializationInfo, as described in Specialization

Constants, and **can** be `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
- 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 VK_SHADER_STAGE_GEOMETRY_BIT, 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 VK_SHADER_STAGE_GEOMETRY_BIT, 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

**Valid Usage (Implicit)**

- sType **must** be VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO
- pNext **must** be NULL
- flags **must** be 0
- stage **must** be a valid VkShaderStageFlagBits value
- module **must** be a valid VkShaderModule handle
- pName **must** be a null-terminated string
- If pSpecializationInfo is not NULL, pSpecializationInfo **must** be a pointer to a valid VkSpecializationInfo structure

## 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*         pCreateInfos,
    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 an array of VkGraphicsPipelineCreateInfo structures.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pPipelines is a pointer to an array 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 given 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 given 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 pointer to an array of `createInfoCount` valid `VkGraphicsPipelineCreateInfo` structures
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pPipelines` **must** be a 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                                 sType;
    const void*                                     pNext;
    VkPipelineCreateFlags                           flags;
    uint32_t                                        stageCount;
    const VkPipelineShaderStageCreateInfo*          pStages;
    const VkPipelineVertexInputStateCreateInfo*     pVertexInputState;
    const VkPipelineInputAssemblyStateCreateInfo*   pInputAssemblyState;
    const VkPipelineTessellationStateCreateInfo*    pTessellationState;
    const VkPipelineViewportStateCreateInfo*        pViewportState;
    const VkPipelineRasterizationStateCreateInfo*   pRasterizationState;
    const VkPipelineMultisampleStateCreateInfo*     pMultisampleState;
    const VkPipelineDepthStencilStateCreateInfo*    pDepthStencilState;
    const VkPipelineColorBlendStateCreateInfo*      pColorBlendState;
    const VkPipelineDynamicStateCreateInfo*         pDynamicState;
    VkPipelineLayout                                layout;
    VkRenderPass                                    renderPass;
    uint32_t                                        subpass;
    VkPipeline                                      basePipelineHandle;
    int32_t                                         basePipelineIndex;
} 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 controlling how the pipeline will be generated, as described below.
- `stageCount` is the number of entries in the `pStages` array.
- `pStages` is an array of size `stageCount` structures of type VkPipelineShaderStageCreateInfo describing the set of the shader stages to be included in the graphics pipeline.
- `pVertexInputState` is a pointer to an instance of the VkPipelineVertexInputStateCreateInfo structure.
- `pInputAssemblyState` is a pointer to an instance of the VkPipelineInputAssemblyStateCreateInfo structure which determines input assembly behavior, as described in Drawing Commands.
- `pTessellationState` is a pointer to an instance of the 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 an instance of the VkPipelineViewportStateCreateInfo structure, and is ignored if the pipeline has rasterization disabled.
- `pRasterizationState` is a pointer to an instance of the VkPipelineRasterizationStateCreateInfo structure.
- `pMultisampleState` is a pointer to an instance of the VkPipelineMultisampleStateCreateInfo, and is ignored if the pipeline has rasterization disabled.
- `pDepthStencilState` is a pointer to an instance of the 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 an instance of the 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 VkPipelineDynamicStateCreateInfo 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.

`pStages` points to an array of VkPipelineShaderStageCreateInfo structures, which were previously described in Compute Pipelines.

Bits which **can** be set in `flags` 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_KHX = 0x00000008,
    VK_PIPELINE_CREATE_DISPATCH_BASE_KHX = 0x00000010,
} 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.
- `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_KHX` specifies that any shader input variables decorated as `DeviceIndex` will be assigned values as if they were decorated as `ViewIndex`.
- `VK_PIPELINE_CREATE_DISPATCH_BASE_KHX` specifies that a compute pipeline **can** be used with

vkCmdDispatchBaseKHX with a non-zero base workgroup.

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.

pDynamicState points to a structure of type VkPipelineDynamicStateCreateInfo.

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 `stage` member of one element of `pStages` **must** be `VK_SHADER_STAGE_VERTEX_BIT`
- The `stage` member of any given 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 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 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 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 format of the attachment does not support color blend operations, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT flag in VkFormatProperties::linearTilingFeatures or VkFormatProperties::optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties
- 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 pointer to an array of pViewportState::viewportCount VkViewport structures
- If no element of the pDynamicStates member of pDynamicState is VK_DYNAMIC_STATE_SCISSOR, the pScissors member of pViewportState **must** be a 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 pointer to a valid VkPipelineViewportStateCreateInfo structure
- If the rasterizerDiscardEnable member of pRasterizationState is VK_FALSE, pMultisampleState **must** be a 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 pointer to a valid `VkPipelineDepthStencilStateCreateInfo` structure
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, and `subpass` uses color attachments, `pColorBlendState` **must** be a 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 `pDepthStencil` is `VK_TRUE`, the `depthBiasClamp` member of `pDepthStencil` **must** be `0.0`
- If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DEPTH_BOUNDS`, and the `depthBoundsTestEnable` member of `pDepthStencil` is `VK_TRUE`, the `minDepthBounds` and `maxDepthBounds` members of `pDepthStencil` **must** be between `0.0` and `1.0`, inclusive
- `layout` **must** be consistent with all shaders specified in `pStages`
- 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 `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_KHX` flag.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of VkPipelineCreateFlagBits values
- `pStages` **must** be a pointer to an array of `stageCount` valid `VkPipelineShaderStageCreateInfo` structures
- `pVertexInputState` **must** be a pointer to a valid `VkPipelineVertexInputStateCreateInfo` structure
- `pInputAssemblyState` **must** be a pointer to a valid `VkPipelineInputAssemblyStateCreateInfo` structure
- `pRasterizationState` **must** be a pointer to a valid `VkPipelineRasterizationStateCreateInfo` structure
- If `pDynamicState` is not `NULL`, `pDynamicState` **must** be a 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 **must** have been created, allocated, or retrieved from the same `VkDevice`

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 an array of VkDynamicState enums which indicate which pieces of pipeline state will use the values from dynamic state commands rather than from the pipeline state creation info.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `pDynamicStates` **must** be a pointer to an array of `dynamicStateCount` valid VkDynamicState values
- `dynamicStateCount` **must** be greater than `0`

The source of difference pieces of dynamic state is determined by the VkPipelineDynamicStateCreateInfo::`pDynamicStates` property of the currently active pipeline, which takes the following 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,
} VkDynamicState;
```

- `VK_DYNAMIC_STATE_VIEWPORT` indicates 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` indicates 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` indicates 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` indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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_DYANMIC_STATE_VIEWPORT_W_SCALING_NV indicates 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 VkPipelineViewportWScalingStateCreateInfo member viewportScalingEnable set to VK_TRUE
- VK_DYNAMIC_STATE_DISCARD_RECTANGLES_EXT indicates 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.

### 9.2.1. Valid Combinations of Stages for Graphics Pipelines

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, the results of fragment processing are undefined. Specifically, any fragment color outputs are considered to have undefined values, and the fragment depth is considered to be unmodified. This **can** be useful for depth-only rendering.

Presence of a shader stage in a pipeline is indicated by including a valid VkPipelineShaderStageCreateInfo 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
    - VkPipelineInputAssemblyStateCreateInfo
    - 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
    - VkPipelineInputAssemblyStateCreateInfo
    - 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

## 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 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 `Vk*PipelineCreateInfo` structure. If this is set, then exactly one of `basePipelineHandle` or `basePipelineIndex` members of the structure **must** have a valid handle/index, and indicates 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 that contains the 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 pointer to a valid `VkPipelineCacheCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pPipelineCache` **must** be a 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 pointer to an array of `initialDataSize` bytes

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 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 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 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 9. 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 VkPipelineCacheHeaderVersion value written as a stream of bytes, with the least significant byte first |
| 8 | 4 | a vendor ID equal to VkPhysicalDeviceProperties::vendorID written as a stream of bytes, with the least significant byte first |
| 12 | 4 | a device ID equal to VkPhysicalDeviceProperties::deviceID written as a stream of bytes, with the least significant byte first |
| 16 | VK_UUID_SIZE | a pipeline cache ID equal to VkPhysicalDeviceProperties::pipelineCacheUUID |

The first four bytes encode the length of the entire pipeline 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. This field is interpreted as a VkPipelineCacheHeaderVersion value, and **must** have one of the following values:

```
typedef enum VkPipelineCacheHeaderVersion {
    VK_PIPELINE_CACHE_HEADER_VERSION_ONE = 1,
} 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 pointer to a `size_t` value
- If the value referenced by `pDataSize` is not `0`, and `pData` is not `NULL`, `pData` **must** be a 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`

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 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` which maps 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 points to a structure of type VkSpecializationMapEntry.

### Valid Usage

- The offset member of any given element of pMapEntries **must** be less than dataSize
- For any given element of pMapEntries, size **must** be less than or equal to dataSize minus offset
- If mapEntryCount is not 0, pMapEntries **must** be a pointer to an array of mapEntryCount valid VkSpecializationMapEntry structures

### Valid Usage (Implicit)

- If dataSize is not 0, pData **must** be a 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 %wgsize 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
%wgsize = 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                             commandBuffer,
    VkPipelineBindPoint                         pipelineBindPoint,
    VkPipeline                                  pipeline);
```

- `commandBuffer` is the command buffer that the pipeline will be bound to.
- `pipelineBindPoint` specifies the bind point, and **must** have one of the values

  ```
  typedef enum VkPipelineBindPoint {
      VK_PIPELINE_BIND_POINT_GRAPHICS = 0,
      VK_PIPELINE_BIND_POINT_COMPUTE = 1,
  } VkPipelineBindPoint;
  ```

  specifying whether `pipeline` will be bound as a compute (`VK_PIPELINE_BIND_POINT_COMPUTE`) or graphics (`VK_PIPELINE_BIND_POINT_GRAPHICS`) pipeline. There are separate bind points for each of graphics and compute, so 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 vkCmdDraw, vkCmdDrawIndexed, vkCmdDrawIndirect, and vkCmdDrawIndexedIndirect. 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

## 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<br>Secondary | Both | Graphics<br>compute | |

# 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. This storage **may** be used for e.g. internal software structures.

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 pointer to an application-defined memory allocation function of type PFN_vkAllocationFunction.
- `pfnReallocation` is a pointer to an application-defined memory reallocation function of type PFN_vkReallocationFunction.
- `pfnFree` is a pointer to an application-defined memory free function of type PFN_vkFreeFunction.
- `pfnInternalAllocation` is a pointer to an application-defined function that is called by the implementation when the implementation makes internal allocations, and it is of type PFN_vkInternalAllocationNotification.
- `pfnInternalFree` is a pointer to an application-defined function that is called by the implementation when the implementation frees internal allocations, and it is of type PFN_vkInternalFreeNotification.

### Valid Usage

- `pfnAllocation` **must** be a pointer to a valid user-defined PFN_vkAllocationFunction
- `pfnReallocation` **must** be a pointer to a valid user-defined PFN_vkReallocationFunction
- `pfnFree` **must** be a 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 the same 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 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 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* which defines its lifetime and which object it is associated with. The allocation scope is provided in the allocationScope parameter passed to callbacks defined in VkAllocationCallbacks. Possible values for this parameter are defined by VkSystemAllocationScope:

```
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,
} VkSystemAllocationScope;
```

- `VK_SYSTEM_ALLOCATION_SCOPE_COMMAND` - The allocation is scoped to the duration of the Vulkan command.
- `VK_SYSTEM_ALLOCATION_SCOPE_OBJECT` - The allocation is scoped to the lifetime of the Vulkan object that is being created or used.
- `VK_SYSTEM_ALLOCATION_SCOPE_CACHE` - The allocation is scoped to the lifetime of a `VkPipelineCache` object.
- `VK_SYSTEM_ALLOCATION_SCOPE_DEVICE` - The allocation is scoped to the lifetime of the Vulkan device.
- `VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE` - 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 an object of type `VkPipelineCache`, the allocator will use the `VK_SYSTEM_ALLOCATION_SCOPE_CACHE` allocation scope. The most specific allocator available is used (pipeline 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,
} VkInternalAllocationType;
```

- `VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE` - 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 an `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 `pPipelineCache`
  - `vkCreateComputePipelines` for `pPipelineCache`
- 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` **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 `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 opaque images that **can** be natively used by the device, or uniform buffer objects that reside in on-device memory.

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` points to an instance of `VkPhysicalDeviceMemoryProperties` structure in which the properties are returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pMemoryProperties` **must** be a pointer to a `VkPhysicalDeviceMemoryProperties` structure 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 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 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_DEVICE_LOCAL_BIT` | `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT`

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`.

The memory types are sorted according to a preorder which serves to aid in easily selecting an appropriate memory type. Given two memory types X and Y, the preorder defines $X \leq Y$ if:

- the memory property bits set for X are a strict subset of the memory property bits set for Y. Or,
- the memory property bits set for X are the same as the memory property bits set for Y, and X uses a memory heap with greater or equal performance (as determined in an implementation-specific manner).

Memory types are ordered in the list such that X is assigned a lesser `memoryTypeIndex` than Y if $(X \leq Y) \wedge \neg (Y \leq X)$ according to the preorder. Note that the list of all allowed memory property flag combinations above satisfies this preorder, but other orders would as well. The goal of this

ordering is to enable applications to use a simple search loop in selecting the proper memory type, along the lines of:

```
// Find a memory type in "memoryTypeBits" that includes all of "properties"
int32_t FindProperties(uint32_t memoryTypeBits, VkMemoryPropertyFlags properties)
{
    for (int32_t i = 0; i < memoryTypeCount; ++i)
    {
        if ((memoryTypeBits & (1 << i)) &&
            ((memoryTypes[i].propertyFlags & properties) == properties))
            return i;
    }
    return -1;
}

// Try to find an optimal memory type, or if it does not exist
// find any compatible memory type
VkMemoryRequirements memoryRequirements;
vkGetImageMemoryRequirements(device, image, &memoryRequirements);
int32_t memoryType = FindProperties(memoryRequirements.memoryTypeBits,
optimalProperties);
if (memoryType == -1)
    memoryType = FindProperties(memoryRequirements.memoryTypeBits,
requiredProperties);
```

The loop will find the first supported memory type that has all bits requested in `properties` set. If there is no exact match, it will find a closest match (i.e. a memory type with the fewest additional bits set), which has some additional bits set but which are not detrimental to the behaviors requested by `properties`. The application **can** first search for the optimal properties, e.g. a memory type that is device-local or supports coherent cached accesses, as appropriate for the intended usage, and if such a memory type is not present **can** fallback to searching for a less optimal but guaranteed set of properties such as "0" or "host-visible and coherent".

To query memory properties, call:

```
void vkGetPhysicalDeviceMemoryProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceMemoryProperties2KHR*       pMemoryProperties);
```

- `physicalDevice` is the handle to the device to query.
- `pMemoryProperties` points to an instance of `VkPhysicalDeviceMemoryProperties2KHR` structure in which the properties are returned.

`vkGetPhysicalDeviceMemoryProperties2KHR` 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 pointer to a `VkPhysicalDeviceMemoryProperties2KHR` structure structure

The `VkPhysicalDeviceMemoryProperties2KHR` structure is defined as:

```
typedef struct VkPhysicalDeviceMemoryProperties2KHR {
    VkStructureType                     sType;
    void*                               pNext;
    VkPhysicalDeviceMemoryProperties    memoryProperties;
} VkPhysicalDeviceMemoryProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `memoryProperties` is a structure of type VkPhysicalDeviceMemoryProperties which is populated with the same values as in vkGetPhysicalDeviceMemoryProperties.

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 attribute flags for the heap. The bits specified in `flags` are:

  ```
  typedef enum VkMemoryHeapFlagBits {
      VK_MEMORY_HEAP_DEVICE_LOCAL_BIT = 0x00000001,
      VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX = 0x00000002,
  } VkMemoryHeapFlagBits;
  ```

  - if `flags` contains `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT`, it means 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.
  - if `flags` contains `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX`, then 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.

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 properties for this memory type. The bits specified in propertyFlags 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,
} VkMemoryPropertyFlagBits;
```

  - if propertyFlags has the VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT bit set, memory allocated with this type is the most efficient for device access. This property will only be set for memory types belonging to heaps with the VK_MEMORY_HEAP_DEVICE_LOCAL_BIT set.
  - if propertyFlags has the VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT bit set, memory allocated with this type **can** be mapped for host access using vkMapMemory.
  - if propertyFlags has the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT bit set, 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.
  - if propertyFlags has the VK_MEMORY_PROPERTY_HOST_CACHED_BIT bit set, 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.
  - if propertyFlags has the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set, 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.

A Vulkan device operates on data in device memory via memory objects that are represented in the API by a VkDeviceMemory handle.

Memory objects are represented by VkDeviceMemory handles:

```
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 an instance of the 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 by 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.

There is an implementation-dependent maximum number of memory allocations which **can** be simultaneously created on a device. This is specified by the `maxMemoryAllocationCount` member of the `VkPhysicalDeviceLimits` structure. If `maxMemoryAllocationCount` is exceeded, `vkAllocateMemory` will return `VK_ERROR_TOO_MANY_OBJECTS`.

*Note*

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` **should** be returned.

### Valid Usage

- The number of currently valid memory objects, allocated from `device`, **must** be less than `VkPhysicalDeviceLimits::maxMemoryAllocationCount`

### Valid Usage (Implicit)

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

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 the memory type index, which selects the properties of the memory to be allocated, as well as the heap the memory will come from.

If the `pNext` chain contains an instance of VkImportMemoryWin32HandleInfoKHX with a non-zero value for `VkImportMemoryWin32HandleInfoKHX::handleType`, or VkImportMemoryFdInfoKHX with a non-zero value for `VkImportMemoryFdInfoKHX::handleType`, the VkMemoryAllocateInfo instance defines a memory import operation. 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_KHX.

## Valid Usage

- `allocationSize` **must** be less than or equal to the amount of memory available to the `VkMemoryHeap` specified by `memoryTypeIndex` and the calling command's `VkDevice`
- `allocationSize` **must** be greater than `0`
- If the `pNext` chain contains an instance of `VkExportMemoryAllocateInfoKHX`, and any of the handle types specified in `VkExportMemoryAllocateInfoKHX`::`handleTypes` require a dedicated allocation, as reported by vkGetPhysicalDeviceImageFormatProperties2KHR in `VkExternalImageFormatPropertiesKHX`::`externalMemoryProperties`::`externalMemoryFeatures` or `VkExternalBufferPropertiesKHX`::`externalMemoryProperties`::`externalMemoryFeatures`, the `pNext` chain **must** contain an instance of `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 VkExportMemoryAllocateInfoKHX, it **must** not contain an instance of VkExportMemoryAllocateInfoNV or VkExportMemoryWin32HandleInfoNV.
- If the `pNext` chain contains an instance of VkImportMemoryWin32HandleInfoKHX, it **must** not contain an instance of VkImportMemoryWin32HandleInfoNV.
- If the parameters define an import operation and the external handle specified was created by the Vulkan API, 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 VkMemoryAllocateFlagsInfoKHX **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 vkGetMemoryWin32HandlePropertiesKHX.
- If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHX`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT_KHX`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT_KHX`, `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_KHX`, `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 vkGetMemoryFdPropertiesKHX.

### 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 or VkMemoryAllocateFlagsInfoKHX
- Each `sType` member in the `pNext` chain **must** be unique

If the `pNext` list 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_TYPE_DEDICATED_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 **must** have been created, allocated, or retrieved from the same `VkDevice`

When allocating memory that **may** be exported to another process or Vulkan instance, add a VkExportMemoryAllocateInfoKHX structure to the `pNext` chain of the VkMemoryAllocateInfo structure, specifying the handle types that **may** be exported.

The VkExportMemoryAllocateInfoKHX structure is defined as:

```
typedef struct VkExportMemoryAllocateInfoKHX {
    VkStructureType                         sType;
    const void*                             pNext;
    VkExternalMemoryHandleTypeFlagsKHX      handleTypes;
} VkExportMemoryAllocateInfoKHX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of VkExternalMemoryHandleTypeFlagBitsKHX 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 VkExternalImageFormatPropertiesKHX or VkExternalBufferPropertiesKHX.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHX`
- `pNext` **must** be `NULL`
- `handleTypes` **must** be a valid combination of VkExternalMemoryHandleTypeFlagBitsKHX values

To specify additional attributes of NT handles exported from a memory object, add the VkExportMemoryWin32HandleInfoKHX structure to the `pNext` chain of the VkMemoryAllocateInfo structure. The `VkExportMemoryWin32HandleInfoKHX` structure is defined as:

```
typedef struct VkExportMemoryWin32HandleInfoKHX {
    VkStructureType                 sType;
    const void*                     pNext;
    const SECURITY_ATTRIBUTES*      pAttributes;
    DWORD                           dwAccess;
    LPCWSTR                         name;
} VkExportMemoryWin32HandleInfoKHX;
```

- `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 UNICODE string to associate with the underlying synchronization primitive 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”[1]. 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_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHX

And

GENERIC_ALL

for handles of the following types:

VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT_KHX
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT_KHX

**1**

https://msdn.microsoft.com/en-us/library/windows/desktop/ms686670.aspx

### Valid Usage

- If VkExportMemoryAllocateInfoKHX::handleTypes does not include VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX, VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHX, VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT_KHX, or VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT_KHX, VkExportMemoryWin32HandleInfoKHX **must** not be in the pNext chain of VkMemoryAllocateInfo.

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHX
- pNext **must** be NULL
- If pAttributes is not NULL, pAttributes **must** be a pointer to a valid SECURITY_ATTRIBUTES value

To import memory created on the same underlying physical device but outside of the current Vulkan instance from a Windows handle, add a VkImportMemoryWin32HandleInfoKHX structure to the pNext chain of the VkMemoryAllocateInfo structure.

The VkImportMemoryWin32HandleInfoKHX structure is defined as:

```
typedef struct VkImportMemoryWin32HandleInfoKHX {
    VkStructureType                          sType;
    const void*                              pNext;
    VkExternalMemoryHandleTypeFlagBitsKHX    handleType;
    HANDLE                                   handle;
} VkImportMemoryWin32HandleInfoKHX;
```

- sType is the type of this structure.

- pNext is NULL or a pointer to an extension-specific structure.
- handleType specifies the type of handle.
- handle is the external handle to import.

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.

### Valid Usage

- If handleType is not 0, it **must** be supported for import, as reported by VkExternalImageFormatPropertiesKHX or VkExternalBufferPropertiesKHX.
- The memory from which handle was exported **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 0, handle **must** be a valid handle of the type specified by handleType.

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHX
- pNext **must** be NULL
- If handleType is not 0, handleType **must** be a valid VkExternalMemoryHandleTypeFlagBitsKHX value

To export a Windows handle representing the underlying resources of a Vulkan device memory object, call:

```
VkResult vkGetMemoryWin32HandleKHX(
    VkDevice                                    device,
    VkDeviceMemory                              memory,
    VkExternalMemoryHandleTypeFlagBitsKHX       handleType,
    HANDLE*                                     pHandle);
```

- device is the logical device that created memory.
- memory is the memory object from which the handle will be exported.
- handleType is the type of handle requested.
- pHandle will return the Windows handle representing the underlying resources of the device memory object.

The properties of the handle returned depend on the value of handleType. See VkExternalMemoryHandleTypeFlagBitsKHX for a description of the properties of the defined external memory handle types.

For handle types defined as NT handles, the handles returned by vkGetMemoryWin32HandleKHX 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

- handleType **must** have been included in VkExportMemoryAllocateInfoKHX::handleTypes when memory was created.
- If handleType is defined as an NT handle, vkGetMemoryWin32HandleKHX **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)

- device **must** be a valid VkDevice handle
- memory **must** be a valid VkDeviceMemory handle
- handleType **must** be a valid VkExternalMemoryHandleTypeFlagBitsKHX value
- pHandle **must** be a 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

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 vkGetMemoryWin32HandlePropertiesKHX(
    VkDevice                                    device,
    VkExternalMemoryHandleTypeFlagBitsKHX       handleType,
    HANDLE                                      handle,
    VkMemoryWin32HandlePropertiesKHX*           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 VkExternalMemoryHandleTypeFlagBitsKHX value
- pMemoryWin32HandleProperties **must** be a pointer to a VkMemoryWin32HandlePropertiesKHX structure

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX

The VkMemoryWin32HandlePropertiesKHX structure returned is defined as:

```
typedef struct VkMemoryWin32HandlePropertiesKHX {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           memoryTypeBits;
} VkMemoryWin32HandlePropertiesKHX;
```

- 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.

To import memory created on the same underlying physical device but outside of the current Vulkan instance from a POSIX file descriptor handle, add a VkImportMemoryFdInfoKHX structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkImportMemoryFdInfoKHX structure is defined as:

```
typedef struct VkImportMemoryFdInfoKHX {
    VkStructureType                           sType;
    const void*                               pNext;
    VkExternalMemoryHandleTypeFlagBitsKHX     handleType;
    int                                       fd;
} VkImportMemoryFdInfoKHX;
```

- `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.

### Valid Usage

- If `handleType` is not `0`, it **must** be supported for import, as reported by VkExternalImageFormatPropertiesKHX or VkExternalBufferPropertiesKHX.
- 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`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHX`
- `pNext` **must** be `NULL`
- If `handleType` is not `0`, `handleType` **must** be a valid VkExternalMemoryHandleTypeFlagBitsKHX value

To export a POSIX file descriptor representing the underlying resources of a Vulkan device memory object, call:

```
VkResult vkGetMemoryFdKHX(
    VkDevice                                    device,
    VkDeviceMemory                              memory,
    VkExternalMemoryHandleTypeFlagBitsKHX       handleType,
    int*                                        pFd);
```

- `device` is the logical device that created `memory`.
- `memory` is the memory object from which the handle will be exported.
- `handleType` is the type of handle requested.
- `pFd` will return a file descriptor representing the underlying resources of the device memory object.

The properties of the file descriptor returned depend on the value of `handleType`. See VkExternalMemoryHandleTypeFlagBitsKHX for a description of the properties of the defined external memory handle types.

Each call to `vkGetMemoryFdKHX` **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.

### Valid Usage

- `handleType` **must** have been included in VkExportMemoryAllocateInfoKHX::`handleTypes` when `memory` was created.
- `handleType` **must** be defined as a POSIX file descriptor handle.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- `handleType` **must** be a valid VkExternalMemoryHandleTypeFlagBitsKHX value
- `pFd` **must** be a pointer to a `int` 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`

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 vkGetMemoryFdPropertiesKHX(
    VkDevice                                    device,
    VkExternalMemoryHandleTypeFlagBitsKHX       handleType,
    int                                         fd,
    VkMemoryFdPropertiesKHX*                    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` will return properties of the handle `fd`.

### Valid Usage

- `fd` **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 VkExternalMemoryHandleTypeFlagBitsKHX value
- `pMemoryFdProperties` **must** be a pointer to a `VkMemoryFdPropertiesKHX` structure

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX`

The `VkMemoryFdPropertiesKHX` structure returned is defined as:

```
typedef struct VkMemoryFdPropertiesKHX {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           memoryTypeBits;
} VkMemoryFdPropertiesKHX;
```

- `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.

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
- pNext **must** be NULL
- 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"[1]. Further, if the structure is not present, the access rights will be

```
code:DXGI_SHARED_RESOURCE_READ | code:DXGI_SHARED_RESOURCE_WRITE
```

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/ms686670.aspx

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_NV`
- `pNext` **must** be `NULL`
- If `pAttributes` is not `NULL`, `pAttributes` **must** be a 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 bit specifying the type of memory handle in `handle`. See VkExternalMemoryHandleTypeFlagBitsNV below for a description of the supported bits.
- `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
- pNext **must** be NULL
- handleType **must** be a valid combination of VkExternalMemoryHandleTypeFlagBitsNV values

Bits which **can** be set in handleType 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,
} VkExternalMemoryHandleTypeFlagBitsNV;
```

- VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV or VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_NV indicate a handle to memory returned by vkGetMemoryWin32HandleNV or, in the case of VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV, one duplicated from such a handle using DuplicateHandle().
- VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_BIT_NV indicates 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 indicates a handle to memory returned by IDXGIResource::GetSharedHandle().

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` points 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 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` list of VkMemoryAllocateInfo includes a `VkMemoryAllocateFlagsInfoKHX` structure, then that structure includes flags and a device mask controlling how many instances of the memory will be allocated.

The `VkMemoryAllocateFlagsInfoKHX` structure is defined as:

```
typedef struct VkMemoryAllocateFlagsInfoKHX {
    VkStructureType             sType;
    const void*                 pNext;
    VkMemoryAllocateFlagsKHX    flags;
    uint32_t                    deviceMask;
} VkMemoryAllocateFlagsInfoKHX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a bitmask of flags controlling the allocation. The bits specified in `flags` are:

  ```
  typedef enum VkMemoryAllocateFlagBitsKHX {
      VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT_KHX = 0x00000001,
  } VkMemoryAllocateFlagBitsKHX;
  ```

  - if `flags` contains `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT_KHX`, memory will be allocated for the devices in `deviceMask`.
- `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_KHX` is set.

If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT_KHX` is not set, the number of instances allocated depends on whether `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX` is set in the memory heap. If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX` 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_KHX` 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 VkPhysicalDeviceGroupPropertiesKHX::`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_KHX` is set, `deviceMask` **must** be a valid device mask.
- If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT_KHX` is set, `deviceMask` **must** not be zero

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO_KHX`
- `flags` **must** be a valid combination of VkMemoryAllocateFlagBitsKHX values

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 queued for execution. The memory **can** remain bound to images or buffers at the time the memory object is freed, but any further use of them (on host or device) for anything other than destroying those objects will result in undefined behavior. 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.

### 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 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` points to a pointer 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`.

It is an application error to call `vkMapMemory` on a memory object that is already 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 mapped for host access, 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 currently be 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 pointer to a pointer
- `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

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.

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 **can** be made available to device access, via availability operations from the VK_ACCESS_HOST_WRITE_BIT access type.

> *Note*
>
> Unmapping non-coherent memory does not implicitly flush the mapped memory, and host writes that have not been flushed **may** not ever be visible to the device.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pMemoryRanges` **must** be a 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 visible to 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.

*Note*

Mapping non-coherent memory does not implicitly invalidate the mapped memory, and device writes that have not been invalidated **must** be made visible before the host reads or overwrites them.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pMemoryRanges` **must** be a 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** currently be 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
- offset **must** be a multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize
- If size is not equal to VK_WHOLE_SIZE, size **must** be a multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize

### 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** currently be 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 pointer to a `VkDeviceSize` value
- `memory` **must** have been created, allocated, or retrieved from `device`

### 10.2.3. 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 vkGetDeviceGroupPeerMemoryFeaturesKHX(
    VkDevice                                    device,
    uint32_t                                    heapIndex,
    uint32_t                                    localDeviceIndex,
    uint32_t                                    remoteDeviceIndex,
    VkPeerMemoryFeatureFlagsKHX*                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 bitmask indicating which types of memory accesses are supported for the combination of heap, local, and remote devices. The bits returned in `pPeerMemoryFeatures` are:

  ```
  typedef enum VkPeerMemoryFeatureFlagBitsKHX {
      VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT_KHX = 0x00000001,
      VK_PEER_MEMORY_FEATURE_COPY_DST_BIT_KHX = 0x00000002,
      VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT_KHX = 0x00000004,
      VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT_KHX = 0x00000008,
  } VkPeerMemoryFeatureFlagBitsKHX;
  ```

  - if `pPeerMemoryFeatures` contains `VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT_KHX`, the memory **can** be accessed as the source of a `vkCmdCopy*` command.
  - if `pPeerMemoryFeatures` contains `VK_PEER_MEMORY_FEATURE_COPY_DST_BIT_KHX`, the memory **can** be accessed as the destination of a `vkCmdCopy*` command.
  - if `pPeerMemoryFeatures` contains `VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT_KHX`, the memory

**can** be read as any other memory access type.

- if `pPeerMemoryFeatures` contains `VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT_KHX`, the memory **can** be written as any other memory access type. Shader atomics are considered to be writes.

`VK_PEER_MEMORY_FEATURE_COPY_DST_BIT_KHX` **must** be supported for all heaps.

If a device doesn't 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.

### 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 pointer to a combination of VkPeerMemoryFeatureFlagBitsKHX values
- `pPeerMemoryFeatures` **must** not be `0`

# 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 an instance of the `VkBufferCreateInfo` structure containing parameters affecting creation of the buffer.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pBuffer` points 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 pointer to a valid `VkBufferCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pBuffer` **must** be a pointer to a `VkBuffer` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

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 describing additional parameters of the buffer. See VkBufferCreateFlagBits below for a description of the supported bits.
- `size` is the size in bytes of the buffer to be created.
- `usage` is a bitmask describing the allowed usages of the buffer. See VkBufferUsageFlagBits below for a description of the supported bits.
- `sharingMode` is the sharing mode of the buffer when it will be accessed by multiple queue families, see VkSharingMode in the Resource Sharing section below for supported values.
- `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`).

Bits which **can** be set in usage 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,
} VkBufferUsageFlagBits;
```

- VK_BUFFER_USAGE_TRANSFER_SRC_BIT indicates 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 indicates that the buffer **can** be used as the destination of a transfer command.
- VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT indicates 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 indicates 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 indicates 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 indicates 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 indicates that the buffer is suitable for passing as the buffer parameter to vkCmdBindIndexBuffer.
- VK_BUFFER_USAGE_VERTEX_BUFFER_BIT indicates that the buffer is suitable for passing as an element of the pBuffers array to vkCmdBindVertexBuffers.
- VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT indicates that the buffer is suitable for passing as the buffer parameter to vkCmdDrawIndirect, vkCmdDrawIndexedIndirect, or vkCmdDispatchIndirect. It is also suitable for passing as the buffer member of VkIndirectCommandsTokenNVX, or sequencesCountBuffer or sequencesIndexBuffer member of VkCmdProcessCommandsInfoNVX

Any combination of bits **can** be specified for usage, but at least one of the bits **must** be set in order to create a valid buffer.

Bits which **can** be set in flags 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,
} VkBufferCreateFlagBits;
```

These bits have the following meanings:

- `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` indicates that the buffer will be backed using sparse memory binding.
- `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` indicates 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` indicates 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.

See Sparse Resource Features and Physical Device Features for details of the sparse memory features supported on a device.

## Valid Usage

- `size` **must** be greater than `0`
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` **must** be a pointer to an array of `queueFamilyIndexCount` `uint32_t` values
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` **must** be greater than `1`
- 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 any of the handle types specified in `VkExternalMemoryImageCreateInfoKHX`::`handleTypes` requires dedicated allocation, as reported by vkGetPhysicalDeviceExternalBufferPropertiesKHX in `VkExternalBufferPropertiesKHX`::`externalMemoryProperties`::`externalMemoryFeatures`, the `pNext` chain **must** contain an instance of `VkDedicatedAllocationBufferCreateInfoNV` with its `dedicatedAllocation` field set to `VK_TRUE`.
- If the `pNext` extension contains an instance of VkExternalMemoryBufferCreateInfoKHX, its `handleTypes` member **must** only contain bits that are also in VkExternalBufferPropertiesKHX::`externalMemoryProperties.pname`:compatibleHandleTypes, as returned by vkGetPhysicalDeviceExternalBufferPropertiesKHX with `pExternalBufferInfo`→`handleType` equal to any one of the handle types specified in VkExternalMemoryBufferCreateInfoKHX::`handleTypes`

## Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkDedicatedAllocationBufferCreateInfoNV
- `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

If the `pNext` list 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 indicates whether the buffer will have a dedicated allocation bound to it.

### Valid Usage

- If dedicatedAllocation is VK_TRUE, VkBufferCreateInfo::flags **must** not include VK_BUFFER_CREATE_SPARSE_BINDING_BIT, VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT, or VK_BUFFER_CREATE_SPARSE_ALIASED_BIT

### 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 VkExternalMemoryBufferCreateInfoKHX structure to the pNext chain of the VkBufferCreateInfo structure. The VkExternalMemoryBufferCreateInfoKHX structure is defined as:

```
typedef struct VkExternalMemoryBufferCreateInfoKHX {
    VkStructureType                       sType;
    const void*                           pNext;
    VkExternalMemoryHandleTypeFlagsKHX    handleTypes;
} VkExternalMemoryBufferCreateInfoKHX;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to an extension-specific structure.
- handleTypes is a bitmask of VkExternalMemoryHandleTypeFlagBitsKHX specifying one or more external memory handle types.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO_KHX`
- `pNext` **must** be `NULL`
- `handleTypes` **must** be a valid combination of VkExternalMemoryHandleTypeFlagBitsKHX values

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 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 an instance of the `VkBufferViewCreateInfo` structure containing parameters to be used to create the buffer.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pView` points 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 pointer to a valid `VkBufferViewCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pView` **must** be a 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 element size of format, then the nearest smaller multiple is used.

### Valid Usage

- `offset` **must** be less than the size of `buffer`
- `offset` **must** be a multiple of `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`
- 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 a multiple of the element size of `format`
- If `range` is not equal to `VK_WHOLE_SIZE`, `range` divided by the element size of `format` **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

### 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

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 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 an instance of the VkImageCreateInfo structure containing parameters to be used to create the image.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pImage points 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 pointer to a valid VkImageCreateInfo structure
- If pAllocator is not NULL, pAllocator **must** be a pointer to a valid VkAllocationCallbacks structure
- pImage **must** be a 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 describing additional parameters of the image. See VkImageCreateFlagBits below for a description of the supported bits.
- `imageType` is a VkImageType specifying the basic dimensionality of the image, as described below. 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 data elements 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 the number of sub-data element samples in the image as defined in VkSampleCountFlagBits. See Multisampling.
- `tiling` is a VkImageTiling specifying the tiling arrangement of the data elements in memory, as described below.
- `usage` is a bitmask describing the intended usage of the image. See VkImageUsageFlagBits below for a description of the supported bits.
- `sharingMode` is the sharing mode of the image when it will be accessed by multiple queue families, and **must** be one of the values described for VkSharingMode in the Resource Sharing section below.
- `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` selects the initial VkImageLayout state 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`

Implementations **may** support additional limits and capabilities beyond those listed above.

To query an implementation's specific capabilities for a given combination of `format`, `type`, `tiling`, `usage`, and `flags`, call vkGetPhysicalDeviceImageFormatProperties2KHR. The return value indicates whether that combination of image settings is supported. On success, the `VkImageFormatProperties` output parameter indicates the set of valid `samples` bits and the limits for `extent`, `mipLevels`, and `arrayLayers`.

To determine the set of valid `usage` bits for a given format, call vkGetPhysicalDeviceFormatProperties.

## Valid Usage

- The combination of `format`, `type`, `tiling`, `usage`, and `flags` **must** be supported, as indicated by a VK_SUCCESS return value from `vkGetPhysicalDeviceImageFormatProperties` invoked with the same values passed to the corresponding parameters.
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` **must** be a pointer to an array of `queueFamilyIndexCount` `uint32_t` values
- If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` **must** be greater than `1`
- `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_CREATE_2D_ARRAY_COMPATIBLE_BIT_KHR`, `imageType` **must** be `VK_IMAGE_TYPE_3D`
- If `imageType` is `VK_IMAGE_TYPE_1D`, `extent.width` **must** be less than or equal to `VkPhysicalDeviceLimits`::`maxImageDimension1D`, or `VkImageFormatProperties`::`maxExtent.width` (as returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure) - whichever is higher
- If `imageType` is `VK_IMAGE_TYPE_2D` and `flags` does not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `extent.width` and `extent.height` **must** be less than or equal to `VkPhysicalDeviceLimits`::`maxImageDimension2D`, or `VkImageFormatProperties`::`maxExtent.width`/height (as returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure) - whichever is higher
- If `imageType` is `VK_IMAGE_TYPE_2D` and `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `extent.width` and `extent.height` **must** be less than or equal to `VkPhysicalDeviceLimits`::`maxImageDimensionCube`, or `VkImageFormatProperties`::`maxExtent.width`/height (as returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure) - whichever is higher
- 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_3D`, `extent.width`, `extent.height` and `extent.depth` **must** be less than or equal to `VkPhysicalDeviceLimits`::`maxImageDimension3D`, or `VkImageFormatProperties`::`maxExtent.width`/height/depth (as returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure) - whichever is higher
- 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 $\log_2(\max(\texttt{extent.width}, \texttt{extent.height}, \texttt{extent.depth})) + 1$.
- If any of `extent.width`, `extent.height`, or `extent.depth` are greater than the equivalently named members of `VkPhysicalDeviceLimits::maxImageDimension3D`, `mipLevels` **must** be less than or equal to `VkImageFormatProperties::maxMipLevels` (as returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure)
- `arrayLayers` **must** be less than or equal to `VkImageFormatProperties::maxArrayLayers` (as returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure)
- If `imageType` is `VK_IMAGE_TYPE_3D`, `arrayLayers` **must** be `1`.
- If `samples` is not `VK_SAMPLE_COUNT_1_BIT`, `imageType` **must** be `VK_IMAGE_TYPE_2D`, `flags` **must** not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`, `tiling` **must** be `VK_IMAGE_TILING_OPTIMAL`, and `mipLevels` **must** be equal to `1`
- 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_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 `VkImageFormatProperties::sampleCounts` returned by `vkGetPhysicalDeviceImageFormatProperties` with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure
- 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 `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 `tiling` is `VK_IMAGE_TILING_LINEAR`, `format` **must** be a format that has at least one supported feature bit present in the value of `VkFormatProperties::linearTilingFeatures` returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`
- If `tiling` is `VK_IMAGE_TILING_LINEAR`, and `VkFormatProperties::linearTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_SAMPLED_BIT`
- If `tiling` is `VK_IMAGE_TILING_LINEAR`, and `VkFormatProperties::linearTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_STORAGE_BIT`
- If `tiling` is `VK_IMAGE_TILING_LINEAR`, and `VkFormatProperties::linearTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`
- If `tiling` is `VK_IMAGE_TILING_LINEAR`, and `VkFormatProperties::linearTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`
- If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, `format` **must** be a format that has at least one supported feature bit present in the value of `VkFormatProperties::optimalTilingFeatures` returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`
- If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, and `VkFormatProperties::optimalTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_SAMPLED_BIT`
- If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, and `VkFormatProperties::optimalTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_STORAGE_BIT`
- If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, and `VkFormatProperties::optimalTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`

- If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, and `VkFormatProperties::optimalTilingFeatures` (as returned by `vkGetPhysicalDeviceFormatProperties` with the same value of `format`) does not include `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`, `usage` **must** not contain `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_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 the `pNext` chain contains an instance of VkExternalMemoryImageCreateInfoNV, it **must** not contain an instance of VkExternalMemoryImageCreateInfoKHX.
- If the `pNext` chain contains an instance of `VkExternalMemoryImageCreateInfoKHX`, and any of the handle types specified in `VkExternalMemoryImageCreateInfoKHX::handleTypes` require a dedicated allocation, as reported by vkGetPhysicalDeviceImageFormatProperties2KHR in `VkExternalImageFormatPropertiesKHX::externalMemoryProperties::externalMemoryFeatures`, the `pNext` chain **must** contain an instance of `VkDedicatedAllocationImageCreateInfoNV` with its `dedicatedAllocation` field set to `VK_TRUE`.
- If the `pNext` chain contains an instance of VkExternalMemoryImageCreateInfoKHX, its `handleTypes` member **must** only contain bits that are also in VkExternalImageFormatPropertiesKHX::`externalMemoryProperties::compatibleHandleTypes`, as returned by vkGetPhysicalDeviceImageFormatProperties2KHR with `format`, `type`, `tiling`, `usage`, and `flags` equal to those in this structure, and with an instance of VkPhysicalDeviceExternalImageFormatInfoKHX in the `pNext` chain, with a `handleType` equal to any one of the handle types specified in VkExternalMemoryImageCreateInfoKHX::`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`, `type`, `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 `flags` contains `VK_IMAGE_CREATE_BIND_SFR_BIT_KHX`, then `mipLevels` **must** be one, `arrayLayers` **must** be one, `imageType` **must** be `VK_IMAGE_TYPE_2D`, and `tiling` **must** be `VK_IMAGE_TILING_OPTIMAL`.
- `initialLayout` **must** be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

### 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 or VkImageSwapchainCreateInfoKHX
- 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` list 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` indicates 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 VkExternalMemoryImageCreateInfoKHX structure to the `pNext` chain of the VkImageCreateInfo structure. The `VkExternalMemoryImageCreateInfoKHX` structure is defined as:

```
typedef struct VkExternalMemoryImageCreateInfoKHX {
    VkStructureType                         sType;
    const void*                             pNext;
    VkExternalMemoryHandleTypeFlagsKHX      handleTypes;
} VkExternalMemoryImageCreateInfoKHX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `handleTypes` is a bitmask of VkExternalMemoryHandleTypeFlagBitsKHX specifying one or more external memory handle types.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_KHX`
- `pNext` **must** be `NULL`
- `handleTypes` **must** be a valid combination of VkExternalMemoryHandleTypeFlagBitsKHX values
- `handleTypes` **must** not be `0`

If the `pNext` list 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`
- `pNext` **must** be `NULL`
- `handleTypes` **must** be a valid combination of VkExternalMemoryHandleTypeFlagBitsNV values

If the `pNext` list of VkImageCreateInfo includes a `VkImageSwapchainCreateInfoKHX` structure, then that structure includes a swapchain handle indicating that the image will be bound to memory from that swapchain.

The `VkImageSwapchainCreateInfoKHX` structure is defined as:

```
typedef struct VkImageSwapchainCreateInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
} VkImageSwapchainCreateInfoKHX;
```

- `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_KHX
- If swapchain is not VK_NULL_HANDLE, swapchain **must** be a valid VkSwapchainKHR handle

The intended usage of an image is specified by the bitmask VkImageCreateInfo::usage. Bits which **can** be set include:

```
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,
} VkImageUsageFlagBits;
```

These bits have the following meanings:

- VK_IMAGE_USAGE_TRANSFER_SRC_BIT indicates that the image **can** be used as the source of a transfer command.
- VK_IMAGE_USAGE_TRANSFER_DST_BIT indicates that the image **can** be used as the destination of a transfer command.
- VK_IMAGE_USAGE_SAMPLED_BIT indicates 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 indicates 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 indicates 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 indicates that the image **can** be used to create a VkImageView suitable for use as a depth/stencil attachment in a VkFramebuffer.
- VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT indicates 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 indicates 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.

Additional parameters of an image are specified by VkImageCreateInfo::flags. Bits which **can** be set

include:

```
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_BIND_SFR_BIT_KHX = 0x00000040,
    VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT_KHR = 0x00000020,
} VkImageCreateFlagBits;
```

These bits have the following meanings:

- `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` indicates that the image will be backed using sparse memory binding.
- `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` indicates 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` indicates 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` indicates that the image **can** be used to create a `VkImageView` with a different format from the image.
- `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` indicates 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_KHR` indicates 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_BIND_SFR_BIT_KHX` indicates that the image **can** be used with a non-zero value of the `SFRRectCount` member of the VkBindImageMemoryInfoKHX structure passed into vkBindImageMemory2KHX. This flag also has the effect of making the image use the standard sparse image block dimensions.

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.

See Sparse Resource Features and Sparse Physical Device Features for more details.

The basic dimensionality of an image is specified by VkImageCreateInfo::`imageType`, which **must** be one of the values

```
typedef enum VkImageType {
    VK_IMAGE_TYPE_1D = 0,
    VK_IMAGE_TYPE_2D = 1,
    VK_IMAGE_TYPE_3D = 2,
} VkImageType;
```

These values specify one-, two-, or three-dimensional images, respectively.

The tiling arrangement of data elements in an image is specified by VkImageCreateInfo::`tiling`, which **must** be one of the values

```
typedef enum VkImageTiling {
    VK_IMAGE_TILING_OPTIMAL = 0,
    VK_IMAGE_TILING_LINEAR = 1,
} VkImageTiling;
```

`VK_IMAGE_TILING_OPTIMAL` specifies optimal tiling (texels are laid out in an implementation-dependent arrangement, for more optimal memory access), and `VK_IMAGE_TILING_LINEAR` specifies linear tiling (texels are laid out in memory in row-major order, possibly with some padding on each row).

To query the host access layout of an image subresource, for an image created with linear tiling, 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` points to a VkSubresourceLayout structure in which the layout is returned.

vkGetImageSubresourceLayout is invariant for the lifetime of a single image.

### Valid Usage

- `image` **must** have been created with `tiling` equal to `VK_IMAGE_TILING_LINEAR`
- The `aspectMask` member of `pSubresource` **must** only have a single bit set

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `image` **must** be a valid `VkImage` handle
- `pSubresource` **must** be a pointer to a valid `VkImageSubresource` structure
- `pLayout` **must** be a 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

- `mipLevel` **must** be less than the `mipLevels` specified in VkImageCreateInfo when the image was created
- `arrayLayer` **must** be less than the `arrayLayers` specified in VkImageCreateInfo when the image was created

### 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 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.

For images created with linear tiling, `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;
```

`arrayPitch` is undefined for images that were not created as arrays. `depthPitch` is defined only for 3D images.

For color formats, the `aspectMask` member of `VkImageSubresource` **must** be `VK_IMAGE_ASPECT_COLOR_BIT`. For depth/stencil formats, `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.

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 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

## 11.4. Image Layouts

Images are stored in implementation-dependent opaque layouts in memory. Implementations **may** support several opaque layouts, and the layout used at any given time is determined by the VkImageLayout state of the image subresource. Each layout has limitations on what kinds of

operations are supported for image subresources using the layout. 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 state 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 images created with `VK_IMAGE_TILING_LINEAR` tiling 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_PRESENT_SRC_KHR = 1000001002,
    VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR = 1000111000,
} VkImageLayout;
```

The type(s) of device access supported by each layout are:

- `VK_IMAGE_LAYOUT_UNDEFINED`: Supports no 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`: Supports no 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 `VK_IMAGE_TILING_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`: **must** only be used as a depth/stencil attachment in a `VkFramebuffer`. This layout is valid only for image subresources of images created with the `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` usage bit enabled.
- `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`: **must** only be used as a read-only depth/stencil attachment in a `VkFramebuffer` and/or 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_DEPTH_STENCIL_ATTACHMENT_BIT` usage bit enabled. Only image subresources of images created with `VK_IMAGE_USAGE_SAMPLED_BIT` **can** be used as sampled image or combined image/sampler in a shader. Similarly, only image subresources of images created with `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` **can** be used as input attachments.
- `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.

For each mechanism of accessing an image in the API, there is a parameter or structure member that controls the image layout used to access the image. 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). 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 the layout specified via the API controlling those accesses.

The image layout of each image subresource **must** be well-defined at each point in the image subresource's lifetime. This means that when performing a layout transition on the 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.

## 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,
} 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 table below. 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 an instance of the `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` points to a `VkImageView` handle in which the resulting image view object is returned.

Some of the image creation parameters are inherited by the view. The remaining parameters are contained in the `pCreateInfo`.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a pointer to a valid `VkImageViewCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pView` **must** be a 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 reserved for future use.
- image is a VkImage on which the view will be created.
- viewType is the type of the image view.
- format is a VkFormat describing the format and type used to interpret data elements in the image.
- components specifies a remapping of color components (or of depth or stencil components after they have been converted into color components). See VkComponentMapping.
- subresourceRange is a VkImageSubresourceRange selecting the set of mipmap levels and array layers to be accessible to the view.

If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, format **can** be different from the image's format, but if they are not equal they **must** be *compatible*. Image format compatibility is defined in the Format Compatibility Classes section.

*Table 10. Image and image view parameter compatibility requirements*

| Dim, Arrayed, MS | Image parameters | View parameters |
|---|---|---|
| | imageType = ci.imageType<br>width = ci.extent.width<br>height = ci.extent.height<br>depth = ci.extent.depth<br>arrayLayers = ci.arrayLayers<br>samples = ci.samples<br>flags = ci.flags<br>where ci is the VkImageCreateInfo used to create image. | baseArrayLayer, layerCount, and levelCount are members of the subresourceRange member. |
| **1D, 0, 0** | imageType = VK_IMAGE_TYPE_1D<br>width ≥ 1<br>height = 1<br>depth = 1<br>arrayLayers ≥ 1<br>samples = 1 | viewType = VK_IMAGE_VIEW_TYPE_1D<br>baseArrayLayer ≥ 0<br>layerCount = 1 |
| **1D, 1, 0** | imageType = VK_IMAGE_TYPE_1D<br>width ≥ 1<br>height = 1<br>depth = 1<br>arrayLayers ≥ 1<br>samples = 1 | viewType = VK_IMAGE_VIEW_TYPE_1D_ARRAY<br>baseArrayLayer ≥ 0<br>layerCount ≥ 1 |
| **2D, 0, 0** | imageType = VK_IMAGE_TYPE_2D<br>width ≥ 1<br>height ≥ 1<br>depth = 1<br>arrayLayers ≥ 1<br>samples = 1 | viewType = VK_IMAGE_VIEW_TYPE_2D<br>baseArrayLayer ≥ 0<br>layerCount = 1 |
| **2D, 1, 0** | imageType = VK_IMAGE_TYPE_2D<br>width ≥ 1<br>height ≥ 1<br>depth = 1<br>arrayLayers ≥ 1<br>samples = 1 | viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY<br>baseArrayLayer ≥ 0<br>layerCount ≥ 1 |
| **2D, 0, 1** | imageType = VK_IMAGE_TYPE_2D<br>width ≥ 1<br>height ≥ 1<br>depth = 1<br>arrayLayers ≥ 1<br>samples > 1 | viewType = VK_IMAGE_VIEW_TYPE_2D<br>baseArrayLayer ≥ 0<br>layerCount = 1 |
| **2D, 1, 1** | imageType = VK_IMAGE_TYPE_2D<br>width ≥ 1<br>height ≥ 1<br>depth = 1<br>arrayLayers ≥ 1<br>samples > 1 | viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY<br>baseArrayLayer ≥ 0<br>layerCount ≥ 1 |

| Dim, Arrayed, MS | Image parameters | View parameters |
|---|---|---|
| **CUBE, 0, 0** | `imageType` = `VK_IMAGE_TYPE_2D`<br>`width` ≥ 1<br>`height` = `width`<br>`depth` = 1<br>`arrayLayers` ≥ 6<br>`samples` = 1<br>`flags` includes `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` | `viewType` = `VK_IMAGE_VIEW_TYPE_CUBE`<br>`baseArrayLayer` ≥ 0<br>`layerCount` = 6 |
| **CUBE, 1, 0** | `imageType` = `VK_IMAGE_TYPE_2D`<br>`width` ≥ 1<br>`height` = width<br>`depth` = 1<br>$N \geq 1$<br>`arrayLayers` ≥ $6 \times N$<br>`samples` = 1<br>`flags` includes `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` | `viewType` = `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`<br>`baseArrayLayer` ≥ 0<br>`layerCount` = $6 \times N$, $N \geq 1$ |
| **3D, 0, 0** | `imageType` = `VK_IMAGE_TYPE_3D`<br>`width` ≥ 1<br>`height` ≥ 1<br>`depth` ≥ 1<br>`arrayLayers` = 1<br>`samples` = 1 | `viewType` = `VK_IMAGE_VIEW_TYPE_3D`<br>`baseArrayLayer` = 0<br>`layerCount` = 1 |
| **3D, 0, 0** | `imageType` = `VK_IMAGE_TYPE_3D`<br>`width` ≥ 1<br>`height` ≥ 1<br>`depth` ≥ 1<br>`arrayLayers` = 1<br>`samples` = 1<br>`flags` includes `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`<br>`flags` does not include VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT | `viewType` = `VK_IMAGE_VIEW_TYPE_2D`<br>`levelCount` = 1<br>`baseArrayLayer` ≥ 0<br>`layerCount` = 1 |

| Dim, Arrayed, MS | Image parameters | View parameters |
|---|---|---|
| **3D, 0, 0** | `imageType` = `VK_IMAGE_TYPE_3D`<br>`width` ≥ 1<br>`height` ≥ 1<br>`depth` ≥ 1<br>`arrayLayers` = 1<br>`samples` = 1<br>`flags` includes `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`<br>`flags` does not include VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT | `viewType` = `VK_IMAGE_VIEW_TYPE_2D_ARRAY`<br>`levelCount` = 1<br>`baseArrayLayer` ≥ 0<br>`layerCount` ≥ 1 |

## 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_KHR set then viewType **must** not be VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY
- If image was created with VK_IMAGE_TILING_LINEAR, format **must** be format that has at least one supported feature bit present in the value of VkFormatProperties::linearTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- 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, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
- If image was created with VK_IMAGE_TILING_LINEAR and usage containing VK_IMAGE_USAGE_SAMPLED_BIT, format **must** be supported for sampled images, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT flag in VkFormatProperties::linearTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- If image was created with VK_IMAGE_TILING_LINEAR and usage containing VK_IMAGE_USAGE_STORAGE_BIT, format **must** be supported for storage images, as specified by the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT flag in VkFormatProperties::linearTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- If image was created with VK_IMAGE_TILING_LINEAR and usage containing VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, format **must** be supported for color attachments, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT flag in VkFormatProperties::linearTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- If image was created with VK_IMAGE_TILING_LINEAR and usage containing VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, format **must** be supported for depth/stencil attachments, as specified by the VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT flag in VkFormatProperties::linearTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- If image was created with VK_IMAGE_TILING_OPTIMAL, format **must** be format that has at least one supported feature bit present in the value of VkFormatProperties::optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- If image was created with VK_IMAGE_TILING_OPTIMAL and usage containing VK_IMAGE_USAGE_SAMPLED_BIT, format **must** be supported for sampled images, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT flag in VkFormatProperties::optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- If image was created with VK_IMAGE_TILING_OPTIMAL and usage containing

VK_IMAGE_USAGE_STORAGE_BIT, format **must** be supported for storage images, as specified by the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT flag in VkFormatProperties::optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format

- If image was created with VK_IMAGE_TILING_OPTIMAL and usage containing VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, format **must** be supported for color attachments, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT flag in VkFormatProperties::optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- If image was created with VK_IMAGE_TILING_OPTIMAL and usage containing VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, format **must** be supported for depth/stencil attachments, as specified by the VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT flag in VkFormatProperties::optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties with the same value of format
- subresourceRange **must** be a valid image subresource range for image (see Image Views)
- 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 not created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, format **must** be identical to the format used to create image
- 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

## Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO
- pNext **must** be NULL
- flags **must** be 0
- 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

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 indicating which aspect(s) of the image are included in the view. See VkImageAspectFlagBits.
- `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, behavior when indexing the last cube is undefined.

`aspectMask` is a bitmask indicating the format being used. Bits which **may** be set include:

```
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,
} VkImageAspectFlagBits;
```

The mask **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. 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 using an imageView 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 imageView 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 imageView 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 components member is of type VkComponentMapping, and 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, and framebuffer attachments.

### Valid Usage

- If levelCount is not VK_REMAINING_MIP_LEVELS, levelCount **must** be non-zero and (baseMipLevel + levelCount) **must** be less than or equal to the mipLevels specified in VkImageCreateInfo when the image was created
- If the imageType specified in VkImageCreateInfo when the image was created was VK_IMAGE_TYPE_3D and the image view is created with the viewType of VkImageViewCreateInfo set to VK_IMAGE_VIEW_TYPE_2D_ARRAY then layerCount **must** be VK_REMAINING_ARRAY_LAYERS, or layerCount **must** be non-zero and (baseArrayLayer + layerCount) **must** be less than or equal to the extent.depth specified in VkImageCreateInfo when the image was created
- If the imageType specified in VkImageCreateInfo when the image was created was not VK_IMAGE_TYPE_3D or the image view is not created with the viewType of VkImageViewCreateInfo set to VK_IMAGE_VIEW_TYPE_2D_ARRAY then layerCount **must** be VK_REMAINING_ARRAY_LAYERS, or layerCount **must** be non-zero and (baseArrayLayer + layerCount) **must** be less than or equal to the arrayLayers specified in VkImageCreateInfo when the image was created

### Valid Usage (Implicit)

- aspectMask **must** be a valid combination of VkImageAspectFlagBits values
- aspectMask **must** not be 0

The VkComponentMapping structure is defined as:

```
typedef struct VkComponentMapping {
    VkComponentSwizzle    r;
    VkComponentSwizzle    g;
    VkComponentSwizzle    b;
    VkComponentSwizzle    a;
} VkComponentMapping;
```

- r determines the component value placed in the R component of the output vector.
- g determines the component value placed in the G component of the output vector.
- b determines the component value placed in the B component of the output vector.

- a determines the component value placed in the A component of the output vector.

Each of r, g, b, and a is one of the values:

```
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,
} VkComponentSwizzle;
```

- VK_COMPONENT_SWIZZLE_IDENTITY: the component is set to the identity swizzle.
- VK_COMPONENT_SWIZZLE_ZERO: the component is set to zero.
- VK_COMPONENT_SWIZZLE_ONE: 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: the component is set to the value of the R component of the image.
- VK_COMPONENT_SWIZZLE_G: the component is set to the value of the G component of the image.
- VK_COMPONENT_SWIZZLE_B: the component is set to the value of the B component of the image.
- VK_COMPONENT_SWIZZLE_A: 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 11. 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 |

### 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

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 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

## 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` points to an instance of the 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 pointer to a `VkMemoryRequirements` structure
- `buffer` **must** have been created, allocated, or retrieved from `device`

To determine the memory requirements for an image resource, 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` points to an instance of the VkMemoryRequirements structure in which the memory requirements of the image object are returned.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `image` **must** be a valid `VkImage` handle
- `pMemoryRequirements` **must** be a 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 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 a `VkImage` that was created with a `VK_IMAGE_TILING_LINEAR` value in the `tiling` member of the `VkImageCreateInfo` structure passed to `vkCreateImage`, 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.

- 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 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, 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.
- 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_BIND_SFR_BIT_KHX bit of the flags member, 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_BIND_SFR_BIT_KHX bit of the flags member, 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 vkGetImageMemoryRequirements::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.

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 VkBindBufferMemoryInfoKHX to vkBindBufferMemory2KHX, with deviceIndexCount equal to zero.

### 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
- If buffer was created with the VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, memoryOffset **must** be a multiple of VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment
- If buffer was created with the VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, memoryOffset **must** be a multiple of VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment
- If buffer was created with the VK_BUFFER_USAGE_STORAGE_BUFFER_BIT, memoryOffset **must** be a multiple of VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment
- 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 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 buffer was not created with VkDedicatedAllocationBufferCreateInfoNV ::dedicatedAllocation equal to VK_TRUE, memory **must** not have been allocated dedicated for a specific buffer or image

### 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`

To attach memory to buffer objects for one or more buffers at a time, call:

```
VkResult vkBindBufferMemory2KHX(
    VkDevice                                    device,
    uint32_t                                    bindInfoCount,
    const VkBindBufferMemoryInfoKHX*            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 structures of type VkBindBufferMemoryInfoKHX, 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 pointer to an array of `bindInfoCount` valid `VkBindBufferMemoryInfoKHX` 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`

`VkBindBufferMemoryInfoKHX` contains members corresponding to the parameters to vkBindBufferMemory, as well as new members to control peer memory bindings.

The `VkBindBufferMemoryInfoKHX` structure is defined as:

```
typedef struct VkBindBufferMemoryInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    VkBuffer           buffer;
    VkDeviceMemory     memory;
    VkDeviceSize       memoryOffset;
    uint32_t           deviceIndexCount;
    const uint32_t*    pDeviceIndices;
} VkBindBufferMemoryInfoKHX;
```

- `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.
- `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 the memory comes from a memory heap with the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX 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 deviceIndexCount is zero and the memory comes from a memory heap without the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX 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

- All valid usage rules from vkBindBufferMemory apply to the identically named members of VkBindBufferMemoryInfoKHX
- 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
- All instances of memory that are bound to **must** have been allocated

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO_KHX
- pNext **must** be NULL
- buffer **must** be a valid VkBuffer handle
- memory **must** be a valid VkDeviceMemory handle
- If deviceIndexCount is not 0, pDeviceIndices **must** be a pointer to an array of deviceIndexCount uint32_t values
- Both of buffer, and memory **must** have been created, allocated, or retrieved from the same VkDevice

To attach memory to an image object, 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 VkBindImageMemoryInfoKHX to vkBindImageMemory2KHX, with deviceIndexCount and SFRRectCount equal to zero.

## 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
- 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 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 image was not created with VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation equal to VK_TRUE, memory **must** not have been allocated dedicated for a specific buffer or image

## 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 vkBindImageMemory2KHX(
    VkDevice                                    device,
    uint32_t                                    bindInfoCount,
    const VkBindImageMemoryInfoKHX*             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 structures of type VkBindImageMemoryInfoKHX, 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 pointer to an array of bindInfoCount valid VkBindImageMemoryInfoKHX 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

VkBindImageMemoryInfoKHX contains members corresponding to the parameters to vkBindImageMemory, as well as new members to control peer memory bindings.

The VkBindImageMemoryInfoKHX structure is defined as:

```
typedef struct VkBindImageMemoryInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    VkImage            image;
    VkDeviceMemory     memory;
    VkDeviceSize       memoryOffset;
    uint32_t           deviceIndexCount;
    const uint32_t*    pDeviceIndices;
    uint32_t           SFRRectCount;
    const VkRect2D*    pSFRRects;
} VkBindImageMemoryInfoKHX;
```

- 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. If SFRRectCount is zero, the number of bytes returned in the VkMemoryRequirements::size member in memory, starting from memoryOffset bytes, will be bound to the specified image.
- deviceIndexCount is the number of elements in pDeviceIndices.
- pDeviceIndices is a pointer to an array of device indices.
- SFRRectCount is the number of elements in pSFRRects.
- pSFRRects is a pointer to an array of rectangles describing which regions of the image are attached to each instance of memory.

If deviceIndexCount is greater than zero, then on device index i the image is attached to the instance of memory on the physical device with device index pDeviceIndices[i].

Let N be the number of physical devices in the logical device. If SFRRectCount is greater than zero, then pSFRRects 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 memory instance j. The blocks of 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 the image, and the block at (0,0) corresponds to memory starting at memoryOffset.

If SFRRectCount and deviceIndexCount are zero and the memory comes from a memory heap with the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX 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 SFRRectCount and deviceIndexCount are zero and the memory comes from a memory heap without

the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX 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

- All valid usage rules from vkBindImageMemory apply to the identically named members of VkBindImageMemoryInfoKHX
- At least one of deviceIndexCount and SFRRectCount **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.
- SFRRectCount **must** either be zero or equal to the number of physical devices in the logical device squared
- If SFRRectCount is not zero, then image **must** have been created with the VK_IMAGE_CREATE_BIND_SFR_BIT_KHX bit set.
- All elements of pSFRRects **must** be valid rectangles contained within the dimensions of the image
- Elements of pSFRRects that correspond to the same instance of the image **must** not overlap and their union **must** cover the entire image.
- For each element of pSFRRects, offset.x **must** be a multiple of the sparse image block width (VkSparseImageFormatProperties::imageGranularity.width) of all non-metadata aspects of the image
- For each element of pSFRRects, extent.width **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
- For each element of pSFRRects, offset.y **must** be a multiple of the sparse image block height (VkSparseImageFormatProperties::imageGranularity.height) of all non-metadata aspects of the image
- For each element of pSFRRects, extent.height **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 height of the image subresource
- All instances of memory that are bound **must** have been allocated
- If image was created with a valid swapchain handle in VkImageSwapchainCreateInfoKHX::swapchain, then the image **must** be bound to memory from that swapchain (using VkBindImageMemorySwapchainInfoKHX).

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO_KHX`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkBindImageMemorySwapchainInfoKHX
- `image` **must** be a valid `VkImage` handle
- `memory` **must** be a valid `VkDeviceMemory` handle
- If `deviceIndexCount` is not `0`, `pDeviceIndices` **must** be a pointer to an array of `deviceIndexCount` `uint32_t` values
- If `SFRRectCount` is not `0`, `pSFRRects` **must** be a pointer to an array of `SFRRectCount` `VkRect2D` structures
- Both of `image`, and `memory` **must** have been created, allocated, or retrieved from the same `VkDevice`

If the `pNext` list of VkBindImageMemoryInfoKHX includes a `VkBindImageMemorySwapchainInfoKHX` structure, then that structure includes a swapchain handle and image index indicating that the image will be bound to memory from that swapchain.

The `VkBindImageMemorySwapchainInfoKHX` structure is defined as:

```
typedef struct VkBindImageMemorySwapchainInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
    uint32_t           imageIndex;
} VkBindImageMemorySwapchainInfoKHX;
```

- `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 `pSFRRects` members of VkBindImageMemoryInfoKHX.

### Valid Usage

- At least one of `swapchain` and VkBindImageMemoryInfoKHX::`memory` **must** be VK_NULL_HANDLE
- `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_KHX`
- `swapchain` **must** be a valid `VkSwapchainKHR` handle

### Host Synchronization

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

*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 require such an additional granularity **may** report a 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 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,
} 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 same queue family at any given time. In order for a different queue family to be able to interpret the memory contents of a range or image subresource, the application **must** perform a queue family ownership transfer.

Upon creation, resources using `VK_SHARING_MODE_EXCLUSIVE` are not owned by any queue family. A buffer or image memory barrier is not required to acquire *ownership* when no queue family owns the resource - it is implicitly acquired upon first use within a queue.

> *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 may 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 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_KHX to represent the external destination or source respectively.

Upon creation, memory objects shared between multiple Vulkan instances or other APIs are not owned by any instance or API. The first entity to access them 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.

## 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, $resource_A$ and $resource_B$, bound respectively to memory $range_A$ and $range_B$. Let $paddedRange_A$ and $paddedRange_B$ be, respectively, $range_A$ and $range_B$ 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 $range_A$ and $range_B$. If one resource is linear and the other is non-linear, then the resources *alias* the memory in the intersection of $paddedRange_A$ and $paddedRange_B$.

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 an opaque, 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 either of two aliases is not host-accessible, then 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 a 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.

Access to resources which alias memory from shaders using variables decorated with `Coherent` are not automatically coherent with each other.

*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.

# 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 an instance of the VkSamplerCreateInfo structure specifying the state of the sampler object.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pSampler points 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 pointer to a valid VkSamplerCreateInfo structure
- If pAllocator is not NULL, pAllocator **must** be a pointer to a valid VkAllocationCallbacks structure
- pSampler **must** be a 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 reserved for future use.
- `magFilter` is the magnification filter to apply to lookups, and is of type:

```
typedef enum VkFilter {
    VK_FILTER_NEAREST = 0,
    VK_FILTER_LINEAR = 1,
    VK_FILTER_CUBIC_IMG = 1000015000,
} VkFilter;
```

- `minFilter` is the minification filter to apply to lookups, and is of type VkFilter.
- `mipmapMode` is the mipmap filter to apply to lookups as described in the Texel Filtering section, and is of type:

```
typedef enum VkSamplerMipmapMode {
    VK_SAMPLER_MIPMAP_MODE_NEAREST = 0,
    VK_SAMPLER_MIPMAP_MODE_LINEAR = 1,
} VkSamplerMipmapMode;
```

- `addressModeU` is the addressing mode for outside [0..1] range for U coordinate. See VkSamplerAddressMode.
- `addressModeV` is the addressing mode for outside [0..1] range for V coordinate. See VkSamplerAddressMode.

- addressModeW is the addressing mode for outside [0..1] range for W coordinate. See VkSamplerAddressMode.
- mipLodBias is the bias to be added to mipmap LOD 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.
- 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 the comparison function to apply to fetched data before filtering as described in the Depth Compare Operation section. See VkCompareOp.
- minLod and maxLod are the values used to clamp the computed level-of-detail value, as described in the Level-of-Detail Operation section. maxLod **must** be greater than or equal to minLod.
- borderColor is the predefined border color to use, as described in the Texel Replacement section, and is of type:

```
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,
} VkBorderColor;
```

- 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, samplers have the following requirements:
  - minFilter and magFilter **must** be equal.
  - mipmapMode **must** be VK_SAMPLER_MIPMAP_MODE_NEAREST.
  - minLod and maxLod **must** be zero.
  - addressModeU and addressModeV **must** each be either VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE or VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER.
  - anisotropyEnable **must** be VK_FALSE.
  - compareEnable **must** be VK_FALSE.
- 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 λ 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.

`addressModeU`, `addressModeV`, and `addressModeW` **must** each have one of the following values:

```
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,
} VkSamplerAddressMode;
```

These values control 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.

- `VK_SAMPLER_ADDRESS_MODE_REPEAT` indicates that the repeat wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT` indicates that the mirrored repeat wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` indicates that the clamp to edge wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER` indicates that the clamp to border wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` indicates that the mirror clamp to edge wrap

mode will be used. This is only valid if the VK_KHR_mirror_clamp_to_edge extension is enabled.

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`
- 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 `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 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_IMG`, `anisotropyEnable` **must** be `VK_FALSE`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `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

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 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

# Chapter 13. Resource Descriptors

Shaders access buffer and image resources by using special shader variables which are indirectly bound to buffer and image views via the API. These variables are organized into sets, where each set of bindings is represented by a *descriptor set* object in the API and a descriptor set is bound all at once. A *descriptor* is an opaque data structure representing a shader resource such as a buffer view, image view, sampler, or combined image sampler. The content of each set is determined by its *descriptor set layout* and the sequence of set layouts that **can** be used by resource variables in shaders within a pipeline is specified in a *pipeline layout*.

Each shader **can** use up to `maxBoundDescriptorSets` (see Limits) descriptor sets, and each descriptor set **can** include bindings for descriptors of all descriptor types. Each shader resource variable is assigned a tuple of (set number, binding number, array element) that defines its location within a descriptor set layout. In GLSL, the set number and binding number are assigned via layout qualifiers, and the array element is implicitly assigned consecutively starting with index equal to zero for the first element of an array (and array element is zero for non-array variables):

*GLSL example*

```
// Assign set number = M, binding number = N, array element = 0
layout (set=M, binding=N) uniform sampler2D variableName;

// Assign set number = M, binding number = N for all array elements, and
// array element = I for the I'th member of the array.
layout (set=M, binding=N) uniform sampler2D variableNameArray[I];
```

*SPIR-V example*

```
// Assign set number = M, binding number = N, array element = 0
                ...
          %1 = OpExtInstImport "GLSL.std.450"
                ...
                OpName %10 "variableName"
                OpDecorate %10 DescriptorSet M
                OpDecorate %10 Binding N
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %6 = OpTypeFloat 32
          %7 = OpTypeImage %6 2D 0 0 0 1 Unknown
          %8 = OpTypeSampledImage %7
          %9 = OpTypePointer UniformConstant %8
         %10 = OpVariable %9 UniformConstant
                ...

// Assign set number = M, binding number = N for all array elements, and
// array element = I for the I'th member of the array.
                ...
          %1 = OpExtInstImport "GLSL.std.450"
                ...
                OpName %13 "variableNameArray"
                OpDecorate %13 DescriptorSet M
                OpDecorate %13 Binding N
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %6 = OpTypeFloat 32
          %7 = OpTypeImage %6 2D 0 0 0 1 Unknown
          %8 = OpTypeSampledImage %7
          %9 = OpTypeInt 32 0
         %10 = OpConstant %9 I
         %11 = OpTypeArray %8 %10
         %12 = OpTypePointer UniformConstant %11
         %13 = OpVariable %12 UniformConstant
                ...
```

## 13.1. Descriptor Types

The following sections outline the various descriptor types supported by Vulkan. Each section defines a descriptor type, and each descriptor type has a manifestation in the shading language and SPIR-V as well as in descriptor sets. There is mostly a one-to-one correspondence between descriptor types and classes of opaque types in the shading language, where the opaque types in the shading language **must** refer to a descriptor in the pipeline layout of the corresponding descriptor type. But there is an exception to this rule as described in Combined Image Sampler.

### 13.1.1. Storage Image

A *storage image* (VK_DESCRIPTOR_TYPE_STORAGE_IMAGE) is a descriptor type that is used for load, store, and atomic operations on image memory from within shaders bound to pipelines.

Loads from storage images do not use samplers and are unfiltered and do not support coordinate wrapping or clamping. Loads are supported in all shader stages for image formats which report support for the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT feature bit via vkGetPhysicalDeviceFormatProperties.

Stores to storage images are supported in compute shaders for image formats which report support for the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT feature.

Storage images also support atomic operations in compute shaders for image formats which report support for the VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT feature.

Load and store operations on storage images **can** only be done on images in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR or VK_IMAGE_LAYOUT_GENERAL layout.

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.

Storage image declarations **must** specify the image format in the shader if the variable is used for atomic operations.

If the shaderStorageImageReadWithoutFormat feature is not enabled, storage image declarations **must** specify the image format in the shader if the variable is used for load operations.

If the shaderStorageImageWriteWithoutFormat feature is not enabled, storage image declarations **must** specify the image format in the shader if the variable is used for store operations.

Storage images are declared in GLSL shader source using uniform image variables of the appropriate dimensionality as well as a format layout qualifier (if necessary):

*GLSL example*

```
layout (set=m, binding=n, r32f) uniform image2D myStorageImage;
```

*SPIR-V example*

```
        ...
%1 = OpExtInstImport "GLSL.std.450"
        ...
        OpName %9 "myStorageImage"
        OpDecorate %9 DescriptorSet m
        OpDecorate %9 Binding n
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 2D 0 0 0 2 R32f
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
        ...
```

### 13.1.2. Sampler

A *sampler* (`VK_DESCRIPTOR_TYPE_SAMPLER`) represents a set of parameters which control address calculations, filtering behavior, and other properties, that **can** be used to perform filtered loads from *sampled images* (see Sampled Image).

Samplers are declared in GLSL shader source using uniform `sampler` variables, where the sampler type has no associated texture dimensionality:

*GLSL Example*

```
layout (set=m, binding=n) uniform sampler mySampler;
```

*SPIR-V example*

```
        ...
%1 = OpExtInstImport "GLSL.std.450"
        ...
        OpName %8 "mySampler"
        OpDecorate %8 DescriptorSet m
        OpDecorate %8 Binding n
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeSampler
%7 = OpTypePointer UniformConstant %6
%8 = OpVariable %7 UniformConstant
        ...
```

### 13.1.3. Sampled Image

A *sampled image* (`VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`) **can** be used (usually in conjunction with a sampler) to retrieve sampled image data. Shaders use a sampled image handle and a sampler

handle to sample data, where the image handle generally defines the shape and format of the memory and the sampler generally defines how coordinate addressing is performed. The same sampler **can** be used to sample from multiple images, and it is possible to sample from the same sampled image with multiple samplers, each containing a different set of sampling parameters.

Sampled images are declared in GLSL shader source using uniform `texture` variables of the appropriate dimensionality:

*GLSL example*

```
layout (set=m, binding=n) uniform texture2D mySampledImage;
```

*SPIR-V example*

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %9 "mySampledImage"
          OpDecorate %9 DescriptorSet m
          OpDecorate %9 Binding n
     %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
          ...
```

### 13.1.4. Combined Image Sampler

A *combined image sampler* (`VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`) represents a sampled image along with a set of sampling parameters. It is logically considered a sampled image and a sampler bound together.

*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.

Combined image samplers are declared in GLSL shader source using uniform `sampler` variables of the appropriate dimensionality:

*GLSL example*

```
layout (set=m, binding=n) uniform sampler2D myCombinedImageSampler;
```

*SPIR-V example*

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %10 "myCombinedImageSampler"
          OpDecorate %10 DescriptorSet m
          OpDecorate %10 Binding n
     %2 = OpTypeVoid
     %3 = OpTypeFunction %2
     %6 = OpTypeFloat 32
     %7 = OpTypeImage %6 2D 0 0 0 1 Unknown
     %8 = OpTypeSampledImage %7
     %9 = OpTypePointer UniformConstant %8
    %10 = OpVariable %9 UniformConstant
          ...
```

`VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` descriptor set entries **can** also be accessed via separate sampler and sampled image shader variables. Such variables refer exclusively to the corresponding half of the descriptor, and **can** be combined in the shader with samplers or sampled images that **can** come from the same descriptor or from other combined or separate descriptor types. There are no additional restrictions on how a separate sampler or sampled image variable is used due to it originating from a combined descriptor.

### 13.1.5. Uniform Texel Buffer

A *uniform texel buffer* (`VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`) represents a tightly packed array of homogeneous formatted data that is stored in a buffer and is made accessible to shaders. Uniform texel buffers are read-only.

Uniform texel buffers are declared in GLSL shader source using uniform `samplerBuffer` variables:

*GLSL example*

```
layout (set=m, binding=n) uniform samplerBuffer myUniformTexelBuffer;
```

*SPIR-V example*

```
          ...
%1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %9 "myUniformTexelBuffer"
          OpDecorate %9 DescriptorSet m
          OpDecorate %9 Binding n
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 Buffer 0 0 0 1 Unknown
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
          ...
```

### 13.1.6. Storage Texel Buffer

A *storage texel buffer* (`VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`) represents a tightly packed array of homogeneous formatted data that is stored in a buffer and is made accessible to shaders. Storage texel buffers differ from uniform texel buffers in that they support stores and atomic operations in shaders, **may** support a different maximum length, and **may** have different performance characteristics.

Storage texel buffers are declared in GLSL shader source using uniform `imageBuffer` variables:

*GLSL example*

```
layout (set=m, binding=n, r32f) uniform imageBuffer myStorageTexelBuffer;
```

*SPIR-V example*

```
          ...
%1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %9 "myStorageTexelBuffer"
          OpDecorate %9 DescriptorSet m
          OpDecorate %9 Binding n
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 Buffer 0 0 0 2 R32f
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
          ...
```

### 13.1.7. Uniform Buffer

A *uniform buffer* (VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER) is a region of structured storage that is made accessible for read-only access to shaders. It is typically used to store medium sized arrays of constants such as shader parameters, matrices and other related data.

Uniform buffers are declared in GLSL shader source using the uniform storage qualifier and block syntax:

*GLSL example*

```
layout (set=m, binding=n) uniform myUniformBuffer
{
    vec4 myElement[32];
};
```

*SPIR-V example*

```
            ...
      %1 = OpExtInstImport "GLSL.std.450"
            ...
            OpName %11 "myUniformBuffer"
            OpMemberName %11 0 "myElement"
            OpName %13 ""
            OpDecorate %10 ArrayStride 16
            OpMemberDecorate %11 0 Offset 0
            OpDecorate %11 Block
            OpDecorate %13 DescriptorSet m
            OpDecorate %13 Binding n
      %2 = OpTypeVoid
      %3 = OpTypeFunction %2
      %6 = OpTypeFloat 32
      %7 = OpTypeVector %6 4
      %8 = OpTypeInt 32 0
      %9 = OpConstant %8 32
     %10 = OpTypeArray %7 %9
     %11 = OpTypeStruct %10
     %12 = OpTypePointer Uniform %11
     %13 = OpVariable %12 Uniform
            ...
```

### 13.1.8. Storage Buffer

A *storage buffer* (VK_DESCRIPTOR_TYPE_STORAGE_BUFFER) is a region of structured storage that supports both read and write access for shaders. In addition to general read and write operations, some members of storage buffers **can** be used as the target of atomic operations. In general, atomic operations are only supported on members that have unsigned integer formats.

Storage buffers are declared in GLSL shader source using buffer storage qualifier and block syntax:

*GLSL example*

```
layout (set=m, binding=n) buffer myStorageBuffer
{
    vec4 myElement[];
};
```

*SPIR-V example*

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %9 "myStorageBuffer"
          OpMemberName %9 0 "myElement"
          OpName %11 ""
          OpDecorate %8 ArrayStride 16
          OpMemberDecorate %9 0 Offset 0
          OpDecorate %9 BufferBlock
          OpDecorate %11 DescriptorSet m
          OpDecorate %11 Binding n
     %2 = OpTypeVoid
     %3 = OpTypeFunction %2
     %6 = OpTypeFloat 32
     %7 = OpTypeVector %6 4
     %8 = OpTypeRuntimeArray %7
     %9 = OpTypeStruct %8
    %10 = OpTypePointer Uniform %9
    %11 = OpVariable %10 Uniform
          ...
```

### 13.1.9. Dynamic Uniform Buffer

A *dynamic uniform buffer* (`VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`) differs from a uniform buffer only in how its address and length are specified. Uniform buffers bind a buffer address and length that is specified in the descriptor set update by a buffer handle, offset and range (see Descriptor Set Updates). With dynamic uniform buffers the buffer handle, offset and range specified in the descriptor set define the base address and length. The dynamic offset which is relative to this base address is taken from the `pDynamicOffsets` parameter to vkCmdBindDescriptorSets (see Descriptor Set Binding). The address used for a dynamic uniform buffer is the sum of the buffer base address and the relative offset. The length is unmodified and remains the range as specified in the descriptor update. The shader syntax is identical for uniform buffers and dynamic uniform buffers.

### 13.1.10. Dynamic Storage Buffer

A *dynamic storage buffer* (`VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`) differs from a storage buffer only in how its address and length are specified. The difference is identical to the difference between uniform buffers and dynamic uniform buffers (see Dynamic Uniform Buffer). The shader

syntax is identical for storage buffers and dynamic storage buffers.

### 13.1.11. Input Attachment

An *input attachment* (`VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`) is an image view that **can** be used for pixel local load operations from within fragment shaders bound to pipelines. Loads from input attachments are unfiltered. 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.

In the shader, input attachments **must** be decorated with their input attachment index in addition to descriptor set and binding numbers.

*GLSL example*

```
layout (input_attachment_index=i, set=m, binding=n) uniform subpassInput
myInputAttachment;
```

*SPIR-V example*

```
     ...
%1 = OpExtInstImport "GLSL.std.450"
     ...
     OpName %9 "myInputAttachment"
     OpDecorate %9 DescriptorSet m
     OpDecorate %9 Binding n
     OpDecorate %9 InputAttachmentIndex i
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 SubpassData 0 0 0 2 Unknown
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
     ...
```

## 13.2. Descriptor Sets

Descriptors are grouped together into descriptor set objects. A descriptor set object is an opaque object that contains 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 hardware 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 an instance of the VkDescriptorSetLayoutCreateInfo structure specifying the state of the descriptor set layout object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pSetLayout` points 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 pointer to a valid `VkDescriptorSetLayoutCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSetLayout` **must** be a 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` provides options for descriptor set layout creation, and is of type VkDescriptorSetLayoutCreateFlags. Bits which **can** be set include:

  ```
  typedef enum VkDescriptorSetLayoutCreateFlagBits {
      VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR = 0x00000001,
  } VkDescriptorSetLayoutCreateFlagBits;
  ```

  If `flags` contains `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`, then descriptor sets **must** not be allocated using this layout, and descriptors are instead pushed via vkCmdPushDescriptorSetKHR.

- `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 the total number of elements of all bindings **must** be less than or equal to VkPhysicalDevicePushDescriptorPropertiesKHR::`maxPushDescriptors`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of VkDescriptorSetLayoutCreateFlagBits values
- If `bindingCount` is not `0`, `pBindings` **must** be a pointer to an array of `bindingCount` valid `VkDescriptorSetLayoutBinding` structures

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. 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. There are no limitations on what combinations of stages **can** be used by 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; later binding a sampler into an immutable sampler slot in a descriptor set is not allowed. If `pImmutableSamplers` is not `NULL`, then it is considered to be a pointer to an array of sampler handles that will be consumed by the set layout and used for the corresponding binding. 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 `descriptorType` is treated as 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 pointer to an array of `descriptorCount` valid `VkSampler` handles
- If `descriptorCount` is not `0`, `stageFlags` **must** be a valid combination of VkShaderStageFlagBits values

### Valid Usage (Implicit)

- `descriptorType` **must** be a valid VkDescriptorType value

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 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 which describes 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 an instance of the VkPipelineLayoutCreateInfo structure specifying the state of the pipeline layout object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pPipelineLayout` points 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 pointer to a valid `VkPipelineLayoutCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pPipelineLayout` **must** be a 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 of the type 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 of the type 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 of the type 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 of the type 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 of the type 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
- 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

## 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 pointer to an array of setLayoutCount valid VkDescriptorSetLayout handles
- If pushConstantRangeCount is not 0, pPushConstantRanges **must** be a pointer to an array of pushConstantRangeCount valid VkPushConstantRange structures

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 result in undefined data being read.
- `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 hardware resources or memory locations within a descriptor set. The assignment of hardware 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 12. Pipeline Layout Resource Limits*

| Total Resources Available | Resource Types |
|---|---|
| `maxDescriptorSetSamplers` | sampler |
| | combined image sampler |
| `maxDescriptorSetSampledImages` | sampled image |
| | combined image sampler |
| | uniform texel buffer |
| `maxDescriptorSetStorageImages` | storage image |
| | storage texel buffer |
| `maxDescriptorSetUniformBuffers` | uniform buffer |
| | uniform buffer dynamic |
| `maxDescriptorSetUniformBuffersDynamic` | uniform buffer dynamic |
| `maxDescriptorSetStorageBuffers` | storage buffer |
| | storage buffer dynamic |
| `maxDescriptorSetStorageBuffersDynamic` | storage buffer dynamic |
| `maxDescriptorSetInputAttachments` | input attachment |

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`

## 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 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 assigns hardware units to resources, the mechanism to assign hardware units 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 an instance of the VkDescriptorPoolCreateInfo structure specifying the state of the descriptor pool object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pDescriptorPool` points 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 pointer to a valid `VkDescriptorPoolCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pDescriptorPool` **must** be a pointer to a `VkDescriptorPool` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

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` specifies certain supported operations on the pool. Bits which **can** be set include:

  ```
  typedef enum VkDescriptorPoolCreateFlagBits {
      VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT = 0x00000001,
  } VkDescriptorPoolCreateFlagBits;
  ```

  If `flags` includes `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT`, then 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.

- `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.

### 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`
- `flags` **must** be a valid combination of VkDescriptorPoolCreateFlagBits values
- `pPoolSizes` **must** be a pointer to an array of `poolSizeCount` valid `VkDescriptorPoolSize` structures
- `poolSizeCount` **must** be greater than `0`

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.

### Valid Usage

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

### 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 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 an instance of the 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 array **must** be at least the length specified by the `descriptorSetCount` member of `pAllocateInfo`.

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. However, the descriptor set **can** be bound in a command buffer without causing errors or exceptions. All entries that are statically used by a pipeline in a drawing or dispatching command **must** have been populated before the descriptor set is bound for use by that command. Entries that are not statically used by a pipeline **can** have uninitialized descriptors or descriptors of resources that have been destroyed, and executing a draw or dispatch with such a descriptor set bound does not cause undefined behavior. This means applications need not populate unused entries with dummy descriptors.

If an allocation fails due to fragmentation, an indeterminate error is returned with an unspecified error code. Any returned error other than `VK_ERROR_OUT_OF_POOL_MEMORY_KHR` or `VK_ERROR_FRAGMENTED_POOL` does not imply its usual meaning: applications **should** assume that the allocation failed due to fragmentation, and create a new descriptor pool.

`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.

> *Note*
>
> Applications **should** check for a negative return value when allocating new descriptor sets, assume that any error other than `VK_ERROR_OUT_OF_POOL_MEMORY_KHR` effectively means `VK_ERROR_FRAGMENTED_POOL`, and try to create a new descriptor pool. If `VK_ERROR_FRAGMENTED_POOL` is the actual return value, it adds certainty to that decision.
>
> The reason for this is that `VK_ERROR_FRAGMENTED_POOL` was only added in a later revision of the 1.0 specification, and so drivers **may** return other errors if they were written against earlier revisions. To ensure full compatibility with earlier patch revisions, these other errors are allowed.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pAllocateInfo` **must** be a pointer to a valid `VkDescriptorSetAllocateInfo` structure
- `pDescriptorSets` **must** be a pointer to an array of `pAllocateInfo`::descriptorSetCount `VkDescriptorSet` handles

## 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_KHR`

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 an array of descriptor set layouts, with each member specifying how the corresponding descriptor set is allocated.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO`
- `pNext` **must** be `NULL`
- `descriptorPool` **must** be a valid `VkDescriptorPool` handle
- `pSetLayouts` **must** be a 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`

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 an array of handles to `VkDescriptorSet` objects.

After a successful call to `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 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`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

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`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

## 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 given 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, that command buffer becomes invalid.

### Valid Usage

- The `dstSet` member of any given element of `pDescriptorWrites` or `pDescriptorCopies` **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 pointer to an array of `descriptorWriteCount` valid `VkWriteDescriptorSet` structures
- If `descriptorCopyCount` is not `0`, `pDescriptorCopies` **must** be a 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.
- `descriptorCount` is the number of descriptors to update (the number of elements in `pImageInfo`, `pBufferInfo`, or `pTexelBufferView`).
- `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` points to an array of VkDescriptorImageInfo structures or is ignored, as described below.
- `pBufferInfo` points to an array of VkDescriptorBufferInfo structures or is ignored, as described below.
- `pTexelBufferView` points 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, 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.

## 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_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 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 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 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 any given 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 any given element of pImageInfo **must** be a valid VkImageView and VkImageLayout, respectively
- If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the offset member of any given 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 any given 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, the buffer member of any given element of pBufferInfo that is non-sparse **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 any given 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 any given 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 any given 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 any given 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 any given 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 any given 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 any given 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 any given element of pImageInfo **must** have been created with VK_IMAGE_USAGE_SAMPLED_BIT set
- If descriptorType is VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, the imageView member of any given 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 any given element of pImageInfo **must** have been created with VK_IMAGE_USAGE_STORAGE_BIT set

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET`
- `pNext` **must** be `NULL`
- `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 **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,
} VkDescriptorType;
```

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`, the elements of the VkWriteDescriptorSet::`pBufferInfo` array of VkDescriptorBufferInfo structures will be used to update the descriptors, and other arrays will be ignored.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, the VkWriteDescriptorSet::`pTexelBufferView` array will be used to update the descriptors, and other arrays will be ignored.

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`, the elements of the VkWriteDescriptorSet::`pImageInfo` array of VkDescriptorImageInfo structures will be used to update the descriptors, and other arrays will be ignored.

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 using `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
- imageLayout **must** match the actual VkImageLayout of each subresource accessible from imageView at the time this descriptor is accessed

### Valid Usage (Implicit)

- Both of imageView, and sampler that are valid handles **must** have been created, allocated, or retrieved from the same VkDevice

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.
- dstSet, dstBinding, and dstArrayElement are the destination set, binding, and array element,

respectively.

- 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.

### 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
- 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

### 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 VkDescriptorUpdateTemplateKHR handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(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, vkUpdateDescriptorSetWithTemplateKHR **can** be used as a replacement for vkUpdateDescriptorSets.

VkDescriptorUpdateTemplateKHR allows implementations to convert a set of descriptor update operations on a single descriptor set to an internal format that, in conjunction with vkUpdateDescriptorSetWithTemplateKHR or vkCmdPushDescriptorSetWithTemplateKHR , **can** be more efficient compared to calling vkUpdateDescriptorSets or vkCmdPushDescriptorSetKHR . The descriptors themselves are not specified in the VkDescriptorUpdateTemplateKHR, rather, offsets into an application provided pointer to host memory are specified, which are combined with a pointer passed to vkUpdateDescriptorSetWithTemplateKHR 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 vkCreateDescriptorUpdateTemplateKHR(
    VkDevice                                    device,
    const VkDescriptorUpdateTemplateCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorUpdateTemplateKHR*              pDescriptorUpdateTemplate);
```

- device is the logical device that creates the descriptor update template.
- pCreateInfo is a pointer to an instance of the VkDescriptorUpdateTemplateCreateInfoKHR structure specifying the set of descriptors to update with a single call to vkCmdPushDescriptorSetWithTemplateKHR or vkUpdateDescriptorSetWithTemplateKHR.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pDescriptorUpdateTemplate points to a VkDescriptorUpdateTemplateKHR 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 pointer to a valid VkDescriptorUpdateTemplateCreateInfoKHR structure
- If pAllocator is not NULL, pAllocator **must** be a pointer to a valid VkAllocationCallbacks structure
- pDescriptorUpdateTemplate **must** be a pointer to a VkDescriptorUpdateTemplateKHR handle

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDescriptorUpdateTemplateCreateInfoKHR structure is defined as:

```
typedef struct VkDescriptorUpdateTemplateCreateInfoKHR {
    VkStructureType                                 sType;
    void*                                           pNext;
    VkDescriptorUpdateTemplateCreateFlagsKHR        flags;
    uint32_t                                        descriptorUpdateEntryCount;
    const VkDescriptorUpdateTemplateEntryKHR*       pDescriptorUpdateEntries;
    VkDescriptorUpdateTemplateTypeKHR               templateType;
    VkDescriptorSetLayout                           descriptorSetLayout;
    VkPipelineBindPoint                             pipelineBindPoint;
    VkPipelineLayout                                pipelineLayout;
    uint32_t                                        set;
} 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 VkDescriptorUpdateTemplateEntryKHR 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_KHR 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_KHR.
- 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_KHR, 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_KHR
- pNext **must** be NULL
- flags **must** be 0
- pDescriptorUpdateEntries **must** be a pointer to an array of descriptorUpdateEntryCount valid VkDescriptorUpdateTemplateEntryKHR structures
- templateType **must** be a valid VkDescriptorUpdateTemplateTypeKHR value
- If descriptorSetLayout is not VK_NULL_HANDLE, descriptorSetLayout **must** be a valid VkDescriptorSetLayout handle
- If pipelineBindPoint is not 0, pipelineBindPoint **must** be a valid VkPipelineBindPoint value
- If pipelineLayout is not VK_NULL_HANDLE, pipelineLayout **must** be a valid VkPipelineLayout handle
- descriptorUpdateEntryCount **must** be greater than 0
- Both of descriptorSetLayout, and pipelineLayout that are valid handles **must** have been created, allocated, or retrieved from the same VkDevice

The descriptor update template type is determined by the VkDescriptorUpdateTemplateCreateInfoKHR::templateType property, which takes the following values:

```
typedef enum VkDescriptorUpdateTemplateTypeKHR {
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET_KHR = 0,
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR = 1,
} VkDescriptorUpdateTemplateTypeKHR;
```

- `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET_KHR` indicates that the descriptor update template will be used for descriptor set updates only.
- `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR` indicates that the descriptor update template will be used for push descriptor updates only.

The `VkDescriptorUpdateTemplateEntryKHR` structure is defined as:

```
typedef struct VkDescriptorUpdateTemplateEntryKHR {
    uint32_t            dstBinding;
    uint32_t            dstArrayElement;
    uint32_t            descriptorCount;
    VkDescriptorType    descriptorType;
    size_t              offset;
    size_t              stride;
} 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`.
- `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.
- `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.
```

### 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

### Valid Usage (Implicit)

- `descriptorType` **must** be a valid VkDescriptorType value

To destroy a descriptor update template, call:

```
void vkDestroyDescriptorUpdateTemplateKHR(
    VkDevice                                    device,
    VkDescriptorUpdateTemplateKHR               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 `VkDescriptorUpdateTemplateKHR` handle
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a 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 `VkDescriptorUpdateTemplateKHR` has been created, descriptor sets **can** be updated by calling:

```
void vkUpdateDescriptorSetWithTemplateKHR(
    VkDevice                                    device,
    VkDescriptorSet                             descriptorSet,
    VkDescriptorUpdateTemplateKHR               descriptorUpdateTemplate,
    const void*                                 pData);
```

- `device` is the logical device that updates the descriptor sets.
- `descriptorSet` is the descriptor set to update
- `descriptorUpdateTemplate` is the `VkDescriptorUpdateTemplateKHR` which specifies the update mapping between the application pointer and the descriptor set to update.
- `pData` is a pointer to memory which contains one or more structures of VkDescriptorImageInfo, VkDescriptorBufferInfo, or VkBufferView used to write the descriptors.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `descriptorSet` **must** be a valid `VkDescriptorSet` handle
- `descriptorUpdateTemplate` **must** be a valid `VkDescriptorUpdateTemplateKHR` handle
- `pData` **must** be a pointer
- `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 VkDescriptorUpdateTemplateEntryKHR 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
    {
        0,                                          // 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
    {
        0,                                          // binding
        3,                                          // dstArrayElement
        1,                                          // descriptorCount
        VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER,    // descriptorType
        offsetof(AppDataStructure, bufferView),     // offset
        sizeof(AppBufferView)                       // stride, bufferViews do not
have to be compact
    },
};

// create an descriptor update template for descriptor set updates
const VkDescriptorUpdateTemplateCreateInfoKHR createInfo =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO_KHR,  // sType
    NULL,                                                          // pNext
    0,                                                             // flags
    3,                                                             //
descriptorUpdateEntryCount
    descriptorUpdateTemplateEntries,                               //
pDescriptorUpdateEntries
```

```
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET_KHR,         // templateType
    myLayout,                                                      //
descriptorSetLayout
    0,                                                             //
pipelineBindPoint, ignored by given templateType
    0,                                                             // pipelineLayout,
ignored by given templateType
    0,                                                             // set, ignored by
given templateType
};

VkDescriptorUpdateTemplateKHR myDescriptorUpdateTemplate;
myResult = vkCreatePipelineLayout(
    myDevice,
    &createInfo,
    NULL,
    &myDescriptorUpdateTemplate);
}


AppDataStructure appData;

// fill appData here or cache it in your engine
vkUpdateDescriptorSetWithTemplateKHR(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                             commandBuffer,
    VkPipelineBindPoint                         pipelineBindPoint,
    VkPipelineLayout                            layout,
    uint32_t                                    firstSet,
    uint32_t                                    descriptorSetCount,
    const VkDescriptorSet*                      pDescriptorSets,
    uint32_t                                    dynamicOffsetCount,
    const uint32_t*                             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 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 length 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 during host execution of the command, or during shader execution of the resulting draws, or any time in between. Thus, the contents **must** not be altered (overwritten by an update command, or freed) between when the command is recorded 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

- Any given 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
- Any given element of `pDynamicOffsets` **must** satisfy the required alignment for the corresponding descriptor binding's descriptor type

## 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 pointer to an array of `descriptorSetCount` valid `VkDescriptorSet` handles
- If `dynamicOffsetCount` is not `0`, `pDynamicOffsets` **must** be a 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<br>Secondary | Both | Graphics<br>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                             commandBuffer,
    VkPipelineBindPoint                         pipelineBindPoint,
    VkPipelineLayout                            layout,
    uint32_t                                    set,
    uint32_t                                    descriptorWriteCount,
    const VkWriteDescriptorSet*                 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 have undefined contents. 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 become invalid.

Valid descriptors **must** be pushed for all bindings that any shaders in a pipeline access, 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. However, if none of the shaders in a pipeline statically use certain bindings in the push descriptor set, then those descriptors need not be valid.

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 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 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<br>Secondary | Both | Graphics<br>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                             commandBuffer,
    VkDescriptorUpdateTemplateKHR               descriptorUpdateTemplate,
    VkPipelineLayout                            layout,
    uint32_t                                    set,
    const void*                                 pData);
```

- `commandBuffer` is the command buffer that the descriptors will be recorded in.
- `descriptorUpdateTemplate` A descriptor update template which defines 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` Points to memory which contains the 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

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `descriptorUpdateTemplate` **must** be a valid `VkDescriptorUpdateTemplateKHR` handle
- `layout` **must** be a valid `VkPipelineLayout` handle
- `pData` **must** be a pointer
- `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<br>Secondary | Both | Graphics<br>compute | |

*API example*

```
struct AppBufferView {
    VkBufferView bufferView;
    uint32_t     applicationRelatedInformation;
};

struct AppDataStructure
{
    VkDescriptorImageInfo  imageInfo;          // a single image info
    // ... some more application related data
};

const VkDescriptorUpdateTemplateEntryKHR 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 an descriptor update template for descriptor set updates
const VkDescriptorUpdateTemplateCreateInfoKHR createInfo =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO_KHR,  // sType
    NULL,                                                          // pNext
    0,                                                             // flags
    1,                                                             //
descriptorUpdateEntryCount
    descriptorUpdateTemplateEntries,                               //
pDescriptorUpdateEntries
    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
};

VkDescriptorUpdateTemplateKHR myDescriptorUpdateTemplate;
myResult = vkCreatePipelineLayout(
    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 an array of `size` bytes containing the new push constant values.

### Valid Usage

- `stageFlags` **must** match exactly the shader stages used in `layout` for the range specified by `offset` and `size`
- `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 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<br>Secondary | Both | Graphics<br>compute | |

# 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 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. 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 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. 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 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 32 or 64 are supported in shader interfaces.

Inputs and outputs of the following types consume a single interface location:

- 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 $n \times m$ 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 a 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.

*Table 13. Shader Input and Output Locations*

| Shader Interface | Locations Available |
|---|---|
| vertex input | `maxVertexInputAttributes` |
| vertex output | `maxVertexOutputComponents` / 4 |
| tessellation control input | `maxTessellationControlPerVertexInputComponents` / 4 |
| tessellation control output | `maxTessellationControlPerVertexOutputComponents` / 4 |
| tessellation evaluation input | `maxTessellationEvaluationInputComponents` / 4 |
| tessellation evaluation output | `maxTessellationEvaluationOutputComponents` / 4 |
| geometry input | `maxGeometryInputComponents` / 4 |
| geometry output | `maxGeometryOutputComponents` / 4 |
| fragment input | `maxFragmentInputComponents` / 4 |
| fragment output | `maxFragmentOutputAttachments` |

### 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 single precision 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 that describes 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 to blending or color attachment writes is 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. 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 Normalized Fixed-Point to Floating-Point; otherwise no type conversion is applied. If the type of the values written by the fragment shader do not match the format of the corresponding color attachment, the result is 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 that describes 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 variable in a push constant block **must** be placed at an `Offset` such that the entire constant value 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 variable in a push constant block **must** not cause the space required for that variable to extend outside the range [0, `maxPushConstantsSize`).

Any variable in 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 `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.

Variables identified with the `UniformConstant` storage class are used only as handles to refer to opaque resources. Such variables **must** be typed as `OpTypeImage`, `OpTypeSampler`, `OpTypeSampledImage`, or arrays of only these types. Variables of type `OpTypeImage` **must** have a `Sampled` operand of 1 (sampled image) or 2 (storage image).

Any array of these types **must** only be indexed with constant integral expressions, except under the following conditions:

- For arrays of `OpTypeImage` variables with `Sampled` operand of 2, if the `shaderStorageImageArrayDynamicIndexing` feature is enabled and the shader module declares the `StorageImageArrayDynamicIndexing` capability, the array **must** only be indexed by dynamically uniform expressions.
- For arrays of `OpTypeSampler`, `OpTypeSampledImage` variables, or `OpTypeImage` variables with `Sampled` operand of 1, if the `shaderSampledImageArrayDynamicIndexing` feature is enabled and the shader module declares the `SampledImageArrayDynamicIndexing` capability, the array **must** only be indexed by dynamically uniform expressions.

The `Sampled Type` of an `OpTypeImage` declaration **must** match the same basic data type as the corresponding resource, or the values obtained by reading or sampling from this image are undefined.

The `Image Format` of an `OpTypeImage` declaration **must** not be **Unknown**, for variables which are used for `OpImageRead` 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`, if the `shaderStorageImageReadWithoutFormat` feature is enabled and the shader module declares the `StorageImageReadWithoutFormat` capability.

Variables identified with the `Uniform` storage class are used to access transparent buffer backed resources. Such variables **must** be:

- typed as `OpTypeStruct`, or arrays of only 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.

Any array of these types **must** only be indexed with constant integral expressions, except under the following conditions.

- For arrays of `Block` variables, if the `shaderUniformBufferArrayDynamicIndexing` feature is enabled and the shader module declares the `UniformBufferArrayDynamicIndexing` capability, the array **must** only be indexed by dynamically uniform expressions.
- For arrays of `BufferBlock` variables, if the `shaderStorageBufferArrayDynamicIndexing` feature is enabled and the shader module declares the `StorageBufferArrayDynamicIndexing` capability, the array **must** only be indexed by dynamically uniform expressions.

The `Offset` decoration for any variable in a `Block` **must** not cause the space required for that variable to extend outside the range [0, `maxUniformBufferRange`). The `Offset` decoration for any variable in a `BufferBlock` **must** not cause the space required for that variable to extend outside the range [0, `maxStorageBufferRange`).

Variables identified with a storage class of `UniformConstant` and a decoration of

InputAttachmentIndex **must** be declared as described in Fragment Input Attachment Interface.

Each shader variable declaration **must** refer to the same type of resource as is indicated by the descriptorType. See Shader Resource and Descriptor Type Correspondence for the relationship between shader declarations and descriptor types.

*Table 14. Shader Resource and Descriptor Type Correspondence*

| Resource type | Descriptor Type |
|---|---|
| sampler | VK_DESCRIPTOR_TYPE_SAMPLER |
| sampled image | VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE |
| storage image | VK_DESCRIPTOR_TYPE_STORAGE_IMAGE |
| combined image sampler | VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER |
| uniform texel buffer | VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER |
| storage texel buffer | VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER |
| uniform buffer | VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER<br>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC |
| storage buffer | VK_DESCRIPTOR_TYPE_STORAGE_BUFFER<br>VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC |
| input attachment | VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT |

*Table 15. Shader Resource and Storage Class Correspondence*

| Resource type | Storage Class | Type | Decoration(s)[1] |
|---|---|---|---|
| sampler | UniformConstant | OpTypeSampler | |
| sampled image | UniformConstant | OpTypeImage (Sampled=1) | |
| storage image | UniformConstant | OpTypeImage (Sampled=2) | |
| combined image sampler | UniformConstant | OpTypeSampledImage | |
| 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) |
| input attachment | UniformConstant | OpTypeImage (Dim =SubpassData, Sampled=2) | InputAttachmentIndex |

**1**

in addition to DescriptorSet and Binding

### 14.5.3. DescriptorSet and Binding Assignment

A variable identified 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.

The range of descriptor sets is between zero and `maxBoundDescriptorSets` minus one. If a descriptor set value is statically used by an entry point there **must** be an associated `pSetLayout` in the corresponding pipeline layout as described in Pipeline Layouts consistency.

If the `Binding` decoration is used with an array, the entire array is identified with that binding value. The size of the array declaration **must** be no larger than the `descriptorCount` of that `VkDescriptorSetLayoutBinding`. 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.

The binding **can** be any 32-bit unsigned integer value, as described in Descriptor Set Layout. Each descriptor set has its own binding name space.

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.

If multiple entry points in the same pipeline refer to the same set and binding, all variable definitions with that `DescriptorSet` and `Binding` **must** have the same basic type.

Not all descriptor sets and bindings specified in a pipeline layout need to be used in a particular shader stage or pipeline, but if a `DescriptorSet` and `Binding` decoration is specified for a variable that is statically used in that shader there **must** be a pipeline layout entry identified with that descriptor set and `binding` and the corresponding `stageFlags` **must** specify the appropriate VkShaderStageFlagBits for that stage.

*Table 16. Shader Resource Limits*

| Resources per Stage | Resource Types |
|---|---|
| `maxPerStageDescriptorSamplers` | sampler |
| | combined image sampler |
| `maxPerStageDescriptorSampledImages` | sampled image |
| | combined image sampler |
| | uniform texel buffer |
| `maxPerStageDescriptorStorageImages` | storage image |
| | storage texel buffer |
| `maxPerStageDescriptorUniformBuffers` | uniform buffer |
| | uniform buffer dynamic |

| Resources per Stage | Resource Types |
|---|---|
| `maxPerStageDescriptorStorageBuffers` | storage buffer |
| | storage buffer dynamic |
| `maxPerStageDescriptorInputAttachments` | input attachment[1] |

**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 `PushConstant` or `Uniform` **must** be explicitly laid out using the `Offset`, `ArrayStride`, and `MatrixStride` decorations. There are two different layouts requirements depending on the specific resources.

**Standard Uniform Buffer Layout**

Member variables of an `OpTypeStruct` with storage class of `Uniform` and a decoration of `Block` (uniform buffers) **must** be laid out according to the following rules.

- The `Offset` Decoration **must** be a multiple of its base alignment, computed recursively as follows:
  - a scalar of size N has a base alignment of N
  - a two-component vector, with components of size N, has a base alignment of 2 N
  - a three- or four-component vector, with components of size N, has a base alignment of 4 N
  - an array has a base alignment equal to the base alignment of its element type, rounded up to a multiple of 16
  - a structure has a base alignment equal to the largest base alignment of any of its members, rounded up to a multiple of 16
  - a row-major matrix of C columns has a base alignment equal to the base alignment of vector of C matrix components
  - a column-major matrix has a base alignment equal to the base alignment of the matrix column type
- Any `ArrayStride` or `MatrixStride` decoration **must** be an integer multiple of the base alignment of the array or matrix from 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 base alignment of that structure or array.
- The numeric order of `Offset` Decorations need not follow member declaration order.

> *Note*
>
> The **std140 layout** in GLSL satisfies these rules.

**Standard Storage Buffer Layout**

Member variables of an `OpTypeStruct` with a storage class of `PushConstant` (push constants), or a

storage class of `Uniform` with a decoration of `BufferBlock` (storage buffers) **must** be laid out as above, except for array and structure base alignment which do not need to be rounded up to a multiple of 16.

> *Note*
>
> The **std430 layout** in GLSL satisfies these rules.

## 14.6. Built-In Variables

Built-in variables are accessed in shaders by declaring a variable decorated with a `BuiltIn` 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.

`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 pixel's center. The K coordinate of the barycentric coordinates **can** be derived given the identity I + J + K = 1.0.

`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 pixel 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 pixel'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 codeBaseVertex **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 $i^{th}$ 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 vertex, fragment, tessellation control, tessellation evaluation, and geometry shaders.

In 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.

`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 vertex, fragment, tessellation control, tessellation evaluation, and geometry shaders.

In 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.

`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 $[0, max(1, physicalDeviceCount))$, where physicalDeviceCount is the `physicalDeviceCount`

member of VkDeviceGroupDeviceCreateInfoKHX.

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 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 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.

`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 the sample corresponding to the shader invocation.

When sample shading is not enabled, the x and y components of `FragCoord` reflect the location of the center of the pixel, (0.5,0.5).

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`

Decorating a variable with the `FragDepth` built-in decoration will make that variable contain the new depth value for all samples covered by the fragment. This value will be used for depth testing and, if the depth test passes, any subsequent write to the depth/stencil attachment.

To write to `FragDepth`, a shader **must** declare the `DepthReplacing` execution mode. If a shader

declares the `DepthReplacing` execution mode and there is an execution path through the shader that does not set `FragDepth`, then the fragment's depth value is undefined for executions of the shader that take that path.

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.

`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.

`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 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.

`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.

`InstanceIndex`

Decorating a variable 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.

`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 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.

The `Layer` decoration **must** be used only within vertex, tessellation evaluation, geometry, and fragment shaders.

In a 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.

`LocalInvocationId`

Decorating a variable with the `LocalInvocationId` built-in decoration will make that variable contain the location of the current 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 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.

`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.

`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 vertex, tessellation control, tessellation evaluation, and geometry shaders.

In a 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 vertex, tessellation control, tessellation evaluation, and geometry shaders.

In a 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 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.

`PrimitiveId`

Decorating a variable with the `PrimitiveId` built-in decoration will make that variable contain the index of the current primitive.

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.

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.

The `PrimitiveId` decoration **must** be used only within fragment, tessellation control, tessellation evaluation, and geometry shaders.

In a tessellation control or tessellation evaluation shader, any variable decorated with `PrimitiveId` **must** be declared using the `Output` storage class.

In a geometry shader, any variable decorated with `PrimitiveId` **must** be declared using either the `Input` or `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 geometry shader, the resulting value is seen through the `PrimitiveId` decorated input variable in the fragment shader.

`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`, per-sample shading is enabled for draws that use that fragment shader.

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 × 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.

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 a fragment shader entry point's interface includes an input variable decorated with `SamplePosition`, per-sample shading is enabled for draws that use that fragment shader.

The `SamplePosition` decoration **must** be used only within fragment shaders.

The variable decorated with `SamplePosition` **must** be declared using the `Input` storage class.

The variable decorated with `SamplePosition` **must** be declared as a two-component vector of 32-

bit floating-point values.

`SubgroupEqMaskKHR`

Decorating a variable with the `SubgroupEqMaskKHR` 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 `SubgroupEqMaskKHR`. All other bits are set to zero.

The variable decorated with `SubgroupEqMaskKHR` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupEqMaskKHR` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupGeMaskKHR`

Decorating a variable with the `SubgroupGeMaskKHR` builtin decoration will make that variable contain the *subgroup mask* of the invocations greater than or equal to the current subgroup invocation. The bits corresponding to the invocations greather than or equal to `SubgroupLocalInvocationId` through `SubgroupSize`-1 are set in the variable decorated with `SubgroupGeMaskKHR`. All other bits are set to zero.

The variable decorated with `SubgroupGeMaskKHR` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupGeMaskKHR` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupGtMaskKHR`

Decorating a variable with the `SubgroupGtMaskKHR` builtin decoration will make that variable contain the *subgroup mask* of the invocations greater than the current subgroup invocation. The bits corresponding to the invocations greater than `SubgroupLocalInvocationId` through `SubgroupSize`-1 are set in the variable decorated with `SubgroupGtMaskKHR`. All other bits are set to zero.

The variable decorated with `SubgroupGtMaskKHR` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupGtMaskKHR` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupLeMaskKHR`

Decorating a variable with the `SubgroupLeMaskKHR` builtin decoration will make that variable contain the *subgroup mask* of the invocations less than or equal to the current subgroup invocation. The bits corresponding to the invocations less than or equal to `SubgroupLocalInvocationId` are set in the variable decorated with `SubgroupLeMaskKHR`. All other bits are set to zero.

The variable decorated with `SubgroupLeMaskKHR` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupLeMaskKHR` **must** be declared as a four-component vector of 32-bit integer values.

`SubgroupLtMaskKHR`

Decorating a variable with the `SubgroupLtMaskKHR` builtin decoration will make that variable

contain the *subgroup mask* of the invocations less than the current subgroup invocation. The bits corresponding to the invocations less than `SubgroupLocalInvocationId` are set in the variable decorated with `SubgroupLtMaskKHR`. All other bits are set to zero.

The variable decorated with `SubgroupLtMaskKHR` **must** be declared using the `Input` storage class.

The variable decorated with `SubgroupLtMaskKHR` **must** be declared as a four-component vector of 32-bit integer values.

`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,`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.

`SubgroupSize`

Decorating a variable with the `SubgroupSize` builtin decoration will make that variable contain the implementation-dependent maximum number of invocations in a subgroup. 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.

`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 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 and 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 vertex, tessellation evaluation, geometry, and fragment shaders.

In a 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 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 (`VkPhysicalDeviceLimits`::`maxViewports` / 32) elements. When a shader writes to this variable, bit B of element M controls whether a primitive is emitted to viewport 32 × M +B. The viewports indicated by the mask are used to select the viewport transform and 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 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 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 and 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.

`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 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 a variable with the `WorkgroupSize` built-in decoration will make that variable 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 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.

# Chapter 15. Image Operations

## 15.1. Image Operations Overview

Image Operations are steps performed by SPIR-V image instructions, where those 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.

SPIR-V Image Instructions include the following functionality:

- `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 level of detail used in the sampling operation based on the coordinates used in neighboring fragments.
  - Instructions with `ExplicitLod` in the name determine the level of detail 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.
- 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), and the first four in that order have the same meanings as unnormalized texel coordinates. 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 the sample index and is taken from the `Sample` image operand.

For all coordinate types, unused coordinates are assigned a value of zero.

*Figure 2. Texel Coordinate Systems*

The Texel Coordinate Systems - For the example shown of an 8x4 texel two dimensional image.

- Normalized texel coordinates:
  - The s coordinate goes from 0.0 to 1.0, left to right.
  - The t coordinate goes from 0.0 to 1.0, top to bottom.
- Unnormalized texel coordinates:
  - The u coordinate goes from -1.0 to 9.0, left to right. The u coordinate within the range 0.0 to 8.0 is within the image, otherwise it is within the border.
  - The v coordinate goes from -1.0 to 5.0, top to bottom. The v coordinate within the range 0.0 to 4.0 is within the image, otherwise it is within the border.
- Integer texel coordinates:
  - The i coordinate goes from -1 to 8, left to right. The i coordinate within the range 0 to 7 addresses texels within the image, otherwise it addresses a border texel.
  - The j coordinate goes from -1 to 5, top to bottom. The j coordinate within the range 0 to 3 addresses texels within the image, otherwise it addresses a border texel.
- 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 weights α and β.

- 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 $i_1j'_1$.

*Figure 3. Texel Coordinate Systems*

The Texel Coordinate Systems - For the example shown of an 8x4 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'.

# 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 ($red_{shared}$, $green_{shared}$, $blue_{shared}$, $exp_{shared}$) as follows:

First, the components (red, green, blue) are clamped to ($red_{clamped}$, $green_{clamped}$, $blue_{clamped}$) as:

$$red_{clamped} = \max(0, \min(sharedexp_{max}, red))$$

$$green_{clamped} = \max(0, \min(sharedexp_{max}, green))$$

$blue_{clamped} = max(0, min(sharedexp_{max}, blue))$

Where:

$$N = 9 \quad \text{number of mantissa bits per component}$$
$$B = 15 \quad \text{exponent bias}$$
$$E_{max} = 31 \quad \text{maximum possible biased exponent value}$$
$$sharedexp_{max} = \frac{(2^N - 1)}{2^N} \times 2^{(E_{max} - B)}$$

> *Note*
>
> 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, $max_{clamped}$ is determined:

$max_{clamped} = max(red_{clamped}, green_{clamped}, blue_{clamped})$

A preliminary shared exponent exp' is computed:

$$exp' = \begin{cases} \lfloor \log_2(max_{clamped}) \rfloor + (B+1) & \text{for } max_{clamped} > 2^{-(B+1)} \\ 0 & \text{for } max_{clamped} \le 2^{-(B+1)} \end{cases}$$

The shared exponent $exp_{shared}$ is computed:

$$max_{shared} = \lfloor \frac{max_{clamped}}{2^{(exp' - B - N)}} + \frac{1}{2} \rfloor$$

$$exp_{shared} = \begin{cases} exp' & \text{for } 0 \le max_{shared} < 2^N \\ exp' + 1 & \text{for } max_{shared} = 2^N \end{cases}$$

Finally, three integer values in the range 0 to $2^N$ are computed:

$$red_{shared} = \lfloor \frac{red_{clamped}}{2^{(exp_{shared} - B - N)}} + \frac{1}{2} \rfloor$$
$$green_{shared} = \lfloor \frac{green_{clamped}}{2^{(exp_{shared} - B - N)}} + \frac{1}{2} \rfloor$$
$$blue_{shared} = \lfloor \frac{blue_{clamped}}{2^{(exp_{shared} - B - N)}} + \frac{1}{2} \rfloor$$

### 15.2.2. Shared Exponent to RGB

A shared exponent color ($red_{shared}$, $green_{shared}$, $blue_{shared}$, $exp_{shared}$) is transformed to an RGB color (red, green, blue) as follows:

$$red = red_{shared} \times 2^{(exp_{shared} - B - N)}$$

$$green = green_{shared} \times 2^{(exp_{shared} - B - N)}$$

$$blue = blue_{shared} \times 2^{(exp_{shared} - B - N)}$$

Where:

N = 9 (number of mantissa bits per component)

B = 15 (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
- Format conversion
- Texel replacement
- Depth comparison
- Conversion to RGBA
- Component swizzle

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.

## 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 Validation

There are a number of cases where a SPIR-V instruction **can** mismatch with the sampler, the image, or both. There are a number of cases where the sampler **can** mismatch with the image. In such cases the value of the texel returned is undefined.

These cases include:

- The sampler `borderColor` is an integer type and the image `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 `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 VkComponentSwizzle for any of the VkComponentMapping components is not `VK_COMPONENT_SWIZZLE_IDENTITY`.
- 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 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 `format` is not one of the depth/stencil formats with a depth component, or the image 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.

#### 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

$0 \le i < w_s$

$0 \le j < h_s$

$0 \le k < d_s$

$0 \le l < layers$

$0 \le n < samples$

where:

$w_s$ = width of the image level

$h_s$ = height of the image level

$d_s$ = depth of the image level

layers = number of layers in the image

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 on the borders along the edges and corners of a cube map image, and Cube map edge handling is performed.

#### Cube Map Edge Handling

If the texel coordinates lie on the borders along the edges and corners of a cube map image, the following steps are performed. Note that this only occurs when using `VK_FILTER_LINEAR` 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 along the border in either only i or only j

  then the texel lies along an edge, so 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 along the border in both i and j

  then the texel lies at a corner and 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, subject to the constraint that if the three available samples have the same value, the replacement texel also has 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.

### 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 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 "KHR_DF_TRANSFER_SRGB" 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 17. Border Color B*

| **Sampler** `borderColor` | **Corresponding Border Color** |
|---|---|
| `VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK` | B = (0.0, 0.0, 0.0, 0.0) |
| `VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK` | B = (0.0, 0.0, 0.0, 1.0) |
| `VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE` | B = (1.0, 1.0, 1.0, 1.0) |
| `VK_BORDER_COLOR_INT_TRANSPARENT_BLACK` | B = (0, 0, 0, 0) |
| `VK_BORDER_COLOR_INT_OPAQUE_BLACK` | B = (0, 0, 0, 1) |
| `VK_BORDER_COLOR_INT_OPAQUE_WHITE` | B = (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 18. 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 | $C_r = B_r$ |
| Two component color format | $C_{rg} = (B_r,B_g)$ |
| Three component color format | $C_{rgb} = (B_r,B_g,B_b)$ |

| Texel Aspect or Format | Component Assignment |
|---|---|
| Four component color format | $C_{rgba} = (B_r,B_g,B_b,B_a)$ |

If the read operation is from a buffer resource, and the `robustBufferAccess` feature is enabled, an invalid texel is replaced as described here.

If the `robustBufferAccess` feature is not enabled, the value of an invalid texel is undefined.

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 read **must** be safe, but 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.

$$D = \begin{cases} 1.0 & \begin{cases} D_{ref} \leq D & \text{for LEQUAL} \\ D_{ref} \geq D & \text{for GEQUAL} \\ D_{ref} < D & \text{for LESS} \\ D_{ref} > D & \text{for GREATER} \\ D_{ref} = D & \text{for EQUAL} \\ D_{ref} \neq D & \text{for NOTEQUAL} \\ true & \text{for ALWAYS} \\ false & \text{for NEVER} \end{cases} \\ 0.0 & \text{otherwise} \end{cases}$$

where, in the depth comparison:

$D_{ref}$ = shaderOp.$D_{ref}$ (from optional SPIR-V operand)

D (texel depth value)

### 15.3.5. Conversion to RGBA

The texel is expanded from one, two, or three to four components based on the image base color:

*Table 19. Texel Color After Conversion To RGBA*

| Texel Aspect or Format | RGBA Color |
|---|---|
| Depth aspect | $C_{rgba}$ = (D,0,0,one) |
| Stencil aspect | $C_{rgba}$ = (S,0,0,one) |
| One component color format | $C_{rgba} = (C_r,0,0,one)$ |
| Two component color format | $C_{rgba} = (C_{rg},0,one)$ |

| Texel Aspect or Format | RGBA Color |
|---|---|
| Three component color format | $C_{rgba} = (C_{rgb}, one)$ |
| Four component color format | $C_{rgba} = C_{rgba}$ |

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. The swizzle **can** rearrange the components of the texel, or substitute zero and one for any components. It is defined as follows for the R component, and operates similarly for the other components.

$$C'_{rgba}[R] = \begin{cases} C_{rgba}[R] & \text{for RED swizzle} \\ C_{rgba}[G] & \text{for GREEN swizzle} \\ C_{rgba}[B] & \text{for BLUE swizzle} \\ C_{rgba}[A] & \text{for ALPHA swizzle} \\ 0 & \text{for ZERO swizzle} \\ one & \text{for ONE swizzle} \\ C_{rgba}[R] & \text{for IDENTITY swizzle} \end{cases}$$

where:

$$\begin{aligned} &C_{rgba}[R] \text{ is the RED component} \\ &C_{rgba}[G] \text{ is the GREEN component} \\ &C_{rgba}[B] \text{ is the BLUE component} \\ &C_{rgba}[A] \text{ is the ALPHA component} \\ &\quad one = 1.0\text{f} && \text{for floating point components} \\ &\quad one = 1 && \text{for integer components} \end{aligned}$$

For each component this is applied to, the `VK_COMPONENT_SWIZZLE_IDENTITY` swizzle selects the corresponding component from $C_{rgba}$.

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.

## 15.4. 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 effect of the write on the image view's memory is undefined, but the write **must** not access memory outside of the image view.

### 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 effect of the write is undefined but **must** be safe. In addition, the write **may** have a side effect that is visible to other image instructions, but **must** not be written to any device memory allocation.

### 15.4.4. Texel Output Format Conversion

Texels 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), using the appropriate equations in 16-Bit Floating-Point Numbers and Fixed-Point Data Conversion.

## 15.5. Derivative Operations

SPIR-V derivative instructions include `OpDPdx`, `OpDPdy`, `OpDPdxFine`, `OpDPdyFine`, `OpDPdxCoarse`, and `OpDPdyCoarse`. Derivative instructions are only available in a fragment shader.

*Figure 4. Implicit Derivatives*

Derivatives are computed as if there is a 2x2 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, therefore derivatives in non-uniform control flow are undefined.

$$
\begin{aligned}
dPdx_{i_1,\ j_0} &= dPdx_{i_0,\ j_0} &&= P_{i_1,\ j_0} - P_{i_0,\ j_0} \\
dPdx_{i_1,\ j_1} &= dPdx_{i_0,\ j_1} &&= P_{i_1,\ j_1} - P_{i_0,\ j_1} \\
dPdy_{i_0,\ j_1} &= dPdy_{i_0,\ j_0} &&= P_{i_0,\ j_1} - P_{i_0,\ j_0} \\
dPdy_{i_1,\ j_1} &= dPdy_{i_1,\ j_0} &&= P_{i_1,\ j_1} - P_{i_1,\ j_0}
\end{aligned}
$$

The `Fine` derivative instructions **must** return the values above, for a group of fragments in a 2x2 neighborhood. Coarse derivatives **may** return only two values. In this case, the values **should** be:

$$dPdx = \begin{cases} dPdx_{i_0,\ j_0} & \text{preferred} \\ dPdx_{i_0,\ j_1} & \end{cases}$$

$$dPdy = \begin{cases} dPdy_{i_0,\ j_0} & \text{preferred} \\ dPdy_{i_1,\ j_0} & \end{cases}$$

`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.

# 15.6. Normalized Texel Coordinate Operations

If the image sampler instruction provides normalized texel coordinates, some of the following operations are performed.

## 15.6.1. Projection Operation

For `Proj` image operations, the 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 level-of-detail 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:

$$\begin{aligned} \partial s / \partial x &= dPdx(s), & \partial s / \partial y &= dPdy(s), & &\text{for 1D, 2D, Cube, or 3D image} \\ \partial t / \partial x &= dPdx(t), & \partial t / \partial y &= dPdy(t), & &\text{for 2D, Cube, or 3D image} \\ \partial u / \partial x &= dPdx(u), & \partial u / \partial y &= dPdy(u), & &\text{for Cube or 3D image} \end{aligned}$$

Partial derivatives not defined above for certain image dimensionalities are set to zero.

For explicit level-of-detail 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.

### 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 20. 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 21. Cube map derivative selection*

| Major Axis Direction | $\partial s_c / \partial x$ | $\partial s_c / \partial y$ | $\partial t_c / \partial x$ | $\partial t_c / \partial y$ | $\partial r_c / \partial x$ | $\partial r_c / \partial y$ |
|---|---|---|---|---|---|---|
| $+r_x$ | $-\partial r_z / \partial x$ | $-\partial r_z / \partial y$ | $-\partial r_y / \partial x$ | $-\partial r_y / \partial y$ | $+\partial r_x / \partial x$ | $+\partial r_x / \partial y$ |
| $-r_x$ | $+\partial r_z / \partial x$ | $+\partial r_z / \partial y$ | $-\partial r_y / \partial x$ | $-\partial r_y / \partial y$ | $-\partial r_x / \partial x$ | $-\partial r_x / \partial y$ |
| $+r_y$ | $+\partial r_x / \partial x$ | $+\partial r_x / \partial y$ | $+\partial r_z / \partial x$ | $+\partial r_z / \partial y$ | $+\partial r_y / \partial x$ | $+\partial r_y / \partial y$ |
| $-r_y$ | $+\partial r_x / \partial x$ | $+\partial r_x / \partial y$ | $-\partial r_z / \partial x$ | $-\partial r_z / \partial y$ | $-\partial r_y / \partial x$ | $-\partial r_y / \partial y$ |
| $+r_z$ | $+\partial r_x / \partial x$ | $+\partial r_x / \partial y$ | $-\partial r_y / \partial x$ | $-\partial r_y / \partial y$ | $+\partial r_z / \partial x$ | $+\partial r_z / \partial y$ |

| Major Axis Direction | $\partial s_c / \partial x$ | $\partial s_c / \partial y$ | $\partial t_c / \partial x$ | $\partial t_c / \partial y$ | $\partial r_c / \partial x$ | $\partial r_c / \partial y$ |
|---|---|---|---|---|---|---|
| $-r_z$ | $-\partial r_x / \partial x$ | $-\partial r_x / \partial y$ | $-\partial r_y / \partial x$ | $-\partial r_y / \partial y$ | $-\partial r_z / \partial x$ | $-\partial r_z / \partial y$ |

## 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

Level-of-detail selection **can** be either explicit (provided explicitly by the image instruction) or implicit (determined from a scale factor calculated from the derivatives).

### 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:

$\partial t/\partial x = \partial t/\partial y = 0$ (for 1D images)

$\partial r/\partial x = \partial r/\partial y = 0$ (for 1D, 2D or Cube images)

and

$w_{base}$ = image.w

$h_{base}$ = image.h

$d_{base}$ = image.d

(for the `baseMipLevel`, from the image descriptor).

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:

$$f_x \text{is continuous and monotonically increasing in each of} m_{ux},\ m_{vx},\ \text{and} m_{wx}$$
$$f_y \text{is continuous and monotonically increasing in each of} m_{uy},\ m_{vy},\ \text{and} m_{wy}$$

$$\max(|m_{ux}|, |m_{vx}|, |m_{wx}|) \le f_x \le \sqrt{2}(|m_{ux}| + |m_{vx}| + |m_{wx}|)$$
$$\max(|m_{uy}|, |m_{vy}|, |m_{wy}|) \le f_y \le \sqrt{2}(|m_{uy}| + |m_{vy}| + |m_{wy}|)$$

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 sampling rate is determined by:

$$N = \min(\lceil \rho_{max} / \rho_{min} \rceil, max_{Aniso})$$

where:

sampler.$max_{Aniso}$ = `maxAnisotropy` (from sampler descriptor)

limits.$max_{Aniso}$ = `maxSamplerAnisotropy` (from physical device limits)

$max_{Aniso}$ = min(sampler.$max_{Aniso}$, limits.$max_{Aniso}$)

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 N **should** be treated as $max_{Aniso}$. However, anytime the footprint is small in texel space the implementation **may** use a smaller value of N, even when $\rho_{min}$ is zero or close to zero.

An implementation **may** round N up to the nearest supported sampling rate.

If N = 1, sampling is isotropic. If N > 1, sampling is anisotropic.

### Level-of-Detail Operation

The *level-of-detail* parameter λ is computed as follows:

$$\lambda_{base}(x, y) = \begin{cases} shaderOp.Lod & \text{(from optional SPIR-V operand)} \\ \log_2\left(\frac{\rho_{max}}{N}\right) & \text{otherwise} \end{cases}$$

$$\lambda'(x, y) = \lambda_{base} + \text{clamp}\left(sampler.bias + shaderOp.bias, -maxSamplerLodBias, maxSamplerLodBias\right)$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \leq \lambda' \leq lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases}$$

where:

$$sampler.bias = mipLodBias \qquad \text{(from sampler descriptor)}$$

$$shaderOp.bias = \begin{cases} Bias & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases}$$

$$sampler.lod_{min} = minLod \qquad \text{(from sampler descriptor)}$$

$$shaderOp.lod_{min} = \begin{cases} MinLod & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases}$$

$$lod_{min} = \max(sampler.lod_{min}, shaderOp.lod_{min})$$

$$lod_{max} = maxLod \qquad \text{(from sampler descriptor)}$$

and maxSamplerLodBias is the value of the VkPhysicalDeviceLimits feature `maxSamplerLodBias`.

### Image Level(s) Selection

The image level(s) d, $d_{hi}$, and $d_{lo}$ which texels are read from are selected based on the level-of-detail parameter, as follows. If the sampler's `mipmapMode` is `VK_SAMPLER_MIPMAP_MODE_NEAREST`, then level d is used:

$$d = \begin{cases} level_{base}, & \lambda \leq 0 \\ nearest(\lambda), & \lambda > 0,\ level_{base} + \lambda \leq q + 0.5 \\ q, & \lambda > 0,\ level_{base} + \lambda > q + 0.5 \end{cases}$$

where:

$$nearest(\lambda) = \begin{cases} \lceil level_{base} + \lambda + 0.5 \rceil - 1, & \text{preferred} \\ \lfloor level_{base} + \lambda + 0.5 \rfloor, & \text{alternative} \end{cases}$$

and

q = `levelCount` - 1

`levelCount` is taken from the `subresourceRange` of the image view.

If the sampler's `mipmapMode` is `VK_SAMPLER_MIPMAP_MODE_LINEAR`, two neighboring levels are selected:

$$d_{hi} = \begin{cases} q, & level_{base} + \lambda \geq q \\ \lfloor level_{base} + \lambda \rfloor, & \text{otherwise} \end{cases}$$
$$d_{lo} = \begin{cases} q, & level_{base} + \lambda \geq q \\ d_{hi} + 1, & \text{otherwise} \end{cases}$$
$$\delta = \lambda - \lfloor \lambda \rfloor$$

δ is the fractional value 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 (d or $d_{hi}$ and $d_{lo}$) used in filtering.

$$u(x, y) = s(x, y) \times width_{level}$$
$$v(x, y) = \begin{cases} 0 & \text{for 1D images} \\ t(x, y) \times height_{level} & \text{otherwise} \end{cases}$$
$$w(x, y) = \begin{cases} 0 & \text{for 2D or Cube images} \\ r(x, y) \times depth_{level} & \text{otherwise} \end{cases}$$
$$a(x, y) = \begin{cases} a(x, y) & \text{for array images} \\ 0 & \text{otherwise} \end{cases}$$

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 = clamp(RNE(a), 0, `layerCount` - 1) + `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:

$$
\begin{aligned}
i &= \lfloor u \rfloor \\
j &= \lfloor v \rfloor \\
k &= \lfloor w \rfloor
\end{aligned}
$$

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 α, β, and γ.

$$
\begin{aligned}
i_0 &= \lfloor u - 0.5 \rfloor \\
i_1 &= i_0 + 1 \\
j_0 &= \lfloor v - 0.5 \rfloor \\
j_1 &= j_0 + 1 \\
k_0 &= \lfloor w - 0.5 \rfloor \\
k_1 &= k_0 + 1 \\
\alpha &= (u - 0.5) - i_0 \\
\beta &= (v - 0.5) - j_0 \\
\gamma &= (w - 0.5) - k_0
\end{aligned}
$$

Cubic filtering (VK_FILTER_CUBIC_IMG) 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 $j_3$, as well as weights α and β.

$$
\begin{aligned}
i_0 &= \lfloor u - \frac{3}{2} \rfloor & i_1 &= i_0 + 1 & i_2 &= i_1 + 1 & i_3 &= i_2 + 1 \\
j_0 &= \lfloor u - \frac{3}{2} \rfloor & j_1 &= j_0 + 1 & j_2 &= j_1 + 1 & j_3 &= j_2 + 1 \\
\alpha &= \operatorname{frac}\left(u - \frac{1}{2}\right) \\
\beta &= \operatorname{frac}\left(v - \frac{1}{2}\right)
\end{aligned}
$$

If the image instruction includes a ConstOffset operand, the constant offsets ($\Delta_i$, $\Delta_j$, $\Delta_k$) are added to (i,j,k) components of the integer texel coordinates.

# 15.8. Image Sample Operations

## 15.8.1. 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.

$$
i = \begin{cases}
i \bmod size & \text{for repeat} \\
(size - 1) - \operatorname{mirror}((i \bmod (2 \times size)) - size) & \text{for mirrored repeat} \\
\operatorname{clamp}(i, 0, size - 1) & \text{for clamp to edge} \\
\operatorname{clamp}(i, -1, size) & \text{for clamp to border} \\
\operatorname{clamp}(\operatorname{mirror}(i), 0, size - 1) & \text{for mirror clamp to edge}
\end{cases}
$$

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.

### 15.8.2. Texel Gathering

SPIR-V instructions with `Gather` in the name return a vector derived from a 2x2 rectangular region of 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:

$$\begin{aligned} \tau[R] &= \tau_{i0j1}[level_{base}][comp] \\ \tau[G] &= \tau_{i1j1}[level_{base}][comp] \\ \tau[B] &= \tau_{i1j0}[level_{base}][comp] \\ \tau[A] &= \tau_{i0j0}[level_{base}][comp] \end{aligned}$$

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

### 15.8.3. Texel Filtering

If λ 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 λ 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.

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 l.

$$\tau[level] = \begin{cases} \tau_{ijk}[level], & \text{for 3D image} \\ \tau_{ij}[level], & \text{for 2D or Cube image} \\ \tau_{i}[level], & \text{for 1D image} \end{cases}$$

Within a mip level, `VK_FILTER_LINEAR` filtering computes a weighted average of 8 (for 3D), 4 (for 2D or Cube), or 2 (for 1D) texel values, using the weights computed earlier:

$$
\begin{aligned}
\tau_{3D}[level] = &(1-\alpha)(1-\beta)(1-\gamma)\tau_{i0\,j0k0}[level] \\
&+(\alpha)(1-\beta)(1-\gamma)\tau_{i1\,j0k0}[level] \\
&+(1-\alpha)(\beta)(1-\gamma)\tau_{i0\,j1k0}[level] \\
&+(\alpha)(\beta)(1-\gamma)\tau_{i1\,j1k0}[level] \\
&+(1-\alpha)(1-\beta)(\gamma)\tau_{i0\,j0k1}[level] \\
&+(\alpha)(1-\beta)(\gamma)\tau_{i1\,j0k1}[level] \\
&+(1-\alpha)(\beta)(\gamma)\tau_{i0\,j1k1}[level] \\
&+(\alpha)(\beta)(\gamma)\tau_{i1\,j1k1}[level]
\end{aligned}
$$

$$
\begin{aligned}
\tau_{2D}[level] = &(1-\alpha)(1-\beta)\tau_{i0\,j0}[level] \\
&+(\alpha)(1-\beta)\tau_{i1\,j0}[level] \\
&+(1-\alpha)(\beta)\tau_{i0\,j1}[level] \\
&+(\alpha)(\beta)\tau_{i1\,j1}[level]
\end{aligned}
$$

$$
\begin{aligned}
\tau_{1D}[level] = &(1-\alpha)\tau_{i0}[level] \\
&+(\alpha)\tau_{i1}[level]
\end{aligned}
$$

$$
\tau[level] = \begin{cases} \tau_{3D}[level], & \text{for 3D image} \\ \tau_{2D}[level], & \text{for 2D or Cube image} \\ \tau_{1D}[level], & \text{for 1D image} \end{cases}
$$

Within a mip level, `VK_FILTER_CUBIC_IMG` filtering computes a weighted average of 16 (for 2D), or 4 (for 1D) texel values, using the weights computed during texel selection.

Catmull-Rom Spine interpolation of four points is defined by the equation:

$$
cinterp(\tau_0, \tau_1, \tau_2, \tau_3, \omega) = \frac{1}{2}\begin{bmatrix} 1 & \omega & \omega^2 & \omega^3 \end{bmatrix} \times \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & 1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \times \begin{bmatrix} \tau_0 \\ \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix}
$$

Using the values calculated in texel selection, this equation is applied to the four points in 1D images. For 2D images, the this equation is evaluated first for each row, and the result is then fed back into the equation and interpolated again:

$\tau_{1D}$[level] = cinterp($\tau_{i0}$[level], $\tau_{i1}$[level], $\tau_{i2}$[level], $\tau_{i3}$[level], α)

$\tau_{j0}$[level] = cinterp($\tau_{i0j0}$[level], $\tau_{i1j0}$[level], $\tau_{i2j0}$[level], $\tau_{i3j0}$[level], α)

$\tau_{j1}$[level] = cinterp($\tau_{i0j1}$[level], $\tau_{i1j1}$[level], $\tau_{i2j1}$[level], $\tau_{i3j1}$[level], α)

$\tau_{j2}$[level] = cinterp($\tau_{i0j2}$[level], $\tau_{i1j2}$[level], $\tau_{i2j2}$[level], $\tau_{i3j2}$[level], α)

$\tau_{j3}$[level] = cinterp($\tau_{i0j3}$[level], $\tau_{i1j3}$[level], $\tau_{i2j3}$[level], $\tau_{i3j3}$[level], α)

$\tau_{2D}$[level] = cinterp($\tau_{j0}$[level], $\tau_{j1}$[level], $\tau_{j2}$[level], $\tau_{j3}$[level], β)

$$
\tau[level] = \begin{cases} \tau_{2D}[level], & \text{for 2D image} \\ \tau_{1D}[level], & \text{for 1D image} \end{cases}
$$

Finally, mipmap filtering either selects a value from one mip level or computes a weighted average

between neighboring mip levels:

$$\tau = \begin{cases} \tau[d], & \text{for mip mode BASE or NEAREST} \\ (1-\delta)\tau[d_{hi}] + \delta\tau[d_{lo}], & \text{for mip mode LINEAR} \end{cases}$$

### 15.8.4. 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 $\tau_{2Daniso}$ is defined using the single isotropic $\tau_{2D}$(u,v) at level d.

$$\tau_{2Daniso} = \frac{1}{N}\sum_{i=1}^{N} \tau_{2D}\left(u\left(x - \frac{1}{2} + \frac{i}{N+1}, y\right), \left(v\left(x - \frac{1}{2} + \frac{i}{N+1}\right), y\right)\right), \qquad \text{when } \rho_x > \rho_y$$

$$\tau_{2Daniso} = \frac{1}{N}\sum_{i=1}^{N} \tau_{2D}\left(u\left(x, y - \frac{1}{2} + \frac{i}{N+1}\right), \left(v\left(x, y - \frac{1}{2} + \frac{i}{N+1}\right)\right)\right), \qquad \text{when } \rho_y \geq \rho_x$$

## 15.9. 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 Filtering: Performed by all `OpImageSample`* and `OpImageSparseSample`* instructions.
- Sparse Residency: Performed by all `OpImageSparse`* instructions.

# Chapter 16. 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.

## 16.1. 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 an instance of the `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 pointer to a valid `VkQueryPoolCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pQueryPool` **must** be a 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 the type of queries managed by the pool, and **must** be one of the values

```
typedef enum VkQueryType {
    VK_QUERY_TYPE_OCCLUSION = 0,
    VK_QUERY_TYPE_PIPELINE_STATISTICS = 1,
    VK_QUERY_TYPE_TIMESTAMP = 2,
} VkQueryType;
```

- queryCount is the number of queries managed by the pool.
- pipelineStatistics is a bitmask indicating 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

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `queryType` **must** be a valid VkQueryType 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 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

## 16.2. Query Operation

The operation of queries is controlled by the commands vkCmdBeginQuery, vkCmdEndQuery, 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 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 is in an undefined state and **must** be reset prior to use. Queries **must** also be reset between uses. Using a query that has not been reset will result in undefined behavior.

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, call:

```
void vkCmdResetQueryPool(
    VkCommandBuffer                                 commandBuffer,
    VkQueryPool                                     queryPool,
    uint32_t                                        firstQuery,
    uint32_t                                        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<br>Secondary | Outside | Graphics<br>compute | |

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 compute commands. Timestamp queries write timestamps 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 indicating constraints on the types of queries that **can** be performed. Bits which **can** be set include:

  ```
  typedef enum VkQueryControlFlagBits {
      VK_QUERY_CONTROL_PRECISE_BIT = 0x00000001,
  } VkQueryControlFlagBits;
  ```

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.

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

- The query identified by `queryPool` and `query` **must** currently not be active
- The query identified by `queryPool` and `query` **must** be unavailable
- 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`
- `queryPool` **must** have been created with a `queryType` that differs from that of any other queries that have been made active, and are currently still active within `commandBuffer`
- `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
- All queries used by the command **must** not be active
- All queries used by the command **must** be unavailable
- If `vkCmdBeginQuery` 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
- `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<br>Secondary | Both | Graphics<br>compute | |

To end a query after the set of desired draw or dispatch commands is executed, call:

```
void vkCmdEndQuery(
    VkCommandBuffer                                 commandBuffer,
    VkQueryPool                                     queryPool,
    uint32_t                                        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.

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

- The query identified by `queryPool` and `query` **must** currently be active
- `query` **must** be less than the number of queries in `queryPool`
- All queries used by the command **must** be active
- 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`

## 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<br>Secondary | Both | Graphics<br>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.
- Each query's result is 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.
- 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. Timestamps write one integer value.
- If more than one query is retrieved and `stride` is not at least as large as the size of the array of integers 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                                    device,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount,
    size_t                                      dataSize,
    void*                                       pData,
    VkDeviceSize                                stride,
    VkQueryResultFlags                          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. `firstQuery` and `queryCount` together define a range of queries. For pipeline statistics queries, each query index in the pool contains one integer value for each bit that is enabled in VkQueryPoolCreateInfo::`pipelineStatistics` when the pool is created.
- `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. Bits which **can** be set include:

  ```
  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,
  } VkQueryResultFlagBits;
  ```

  - `VK_QUERY_RESULT_64_BIT` indicates 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` indicates that Vulkan will wait for each query's status to become available before retrieving its results.
  - `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` indicates that the availability status accompanies the results.
  - `VK_QUERY_RESULT_PARTIAL_BIT` indicates that returning partial results is acceptable.

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 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 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.

VK_QUERY_RESULT_PARTIAL_BIT **must** not be used if the pool's queryType is VK_QUERY_TYPE_TIMESTAMP.

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` 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`
- 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`

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `queryPool` **must** be a valid `VkQueryPool` handle
- `pData` **must** be a 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`

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.

If no bits are set in `flags`, results for all requested queries in the available state are written as 32-bit unsigned integer values, and nothing is written for queries in the unavailable state.

If `VK_QUERY_RESULT_64_BIT` is set, the results are written as an array of 64-bit unsigned integer values as described for vkGetQueryPoolResults.

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

### 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<br>Secondary | Outside | Graphics<br>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.

## 16.3. 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.

## 16.4. 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. Further, 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, 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.

The pipeline statistic counters are individually enabled for query pools with `VkQueryPoolCreateInfo::pipelineStatistics`, and for secondary command buffers with `VkCommandBufferInheritanceInfo::pipelineStatistics`.

Bits which **can** be set in `pipelineStatistics` include:

```
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,
} VkQueryPipelineStatisticFlagBits;
```

These bits have the following meanings:

- If `VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT` is set, 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.

- If `VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT` is set, 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.
- If `VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT` is set, 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`).

## 16.5. 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                             commandBuffer,
    VkPipelineStageFlagBits                     pipelineStage,
    VkQueryPool                                 queryPool,
    uint32_t                                    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 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.

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

- 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 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<br>Secondary | Both | Graphics<br>compute | Transfer |

# Chapter 17. Clear Commands

## 17.1. 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                                 commandBuffer,
    VkImage                                         image,
    VkImageLayout                                   imageLayout,
    const VkClearColorValue*                        pColor,
    uint32_t                                        rangeCount,
    const VkImageSubresourceRange*                  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 that contains the values the image subresource ranges will be cleared to (see Clear Values below).
- `rangeCount` is the number of image subresource range structures in `pRanges`.
- `pRanges` points to an array of VkImageSubresourceRange structures that describe a range of mipmap levels, array layers, and aspects to be cleared, as described in Image Views. The `aspectMask` of all image subresource ranges **must** only include `VK_IMAGE_ASPECT_COLOR_BIT`.

Each specified range in `pRanges` is cleared to the value specified by `pColor`.

## Valid Usage

- `image` **must** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by `vkGetPhysicalDeviceFormatProperties`
- `image` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- 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_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- The image range of any given element of `pRanges` **must** be an image subresource range that is contained within `image`
- `image` **must** not have a compressed or depth/stencil format

## 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 pointer to a valid `VkClearColorValue` union
- `pRanges` **must** be a 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<br>Secondary | Outside | Graphics<br>compute | Transfer |

To clear one or more subranges of a depth/stencil image, call:

```
void vkCmdClearDepthStencilImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     image,
    VkImageLayout                               imageLayout,
    const VkClearDepthStencilValue*             pDepthStencil,
    uint32_t                                    rangeCount,
    const VkImageSubresourceRange*              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 that contains the values 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` points to an array of VkImageSubresourceRange structures that describe a range of mipmap levels, array layers, and aspects to be cleared, as described in Image Views. The `aspectMask` of each image subresource range in `pRanges` **can** include `VK_IMAGE_ASPECT_DEPTH_BIT` if the image format has a depth component, and `VK_IMAGE_ASPECT_STENCIL_BIT` if the image format has a stencil component. `pDepthStencil` is a pointer to a `VkClearDepthStencilValue` structure that contains the values the image subresource ranges will be cleared to (see Clear Values below).

## Valid Usage

- `image` **must** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by `vkGetPhysicalDeviceFormatProperties`
- `image` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- 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 image range of any given element of `pRanges` **must** be an image subresource range that is contained within `image`
- `image` **must** have a depth/stencil format

## 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 pointer to a valid `VkClearDepthStencilValue` structure
- `pRanges` **must** be a 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<br>Secondary | Outside | Graphics | Transfer |

Clears outside render pass instances are treated as transfer operations for the purposes of memory barriers.

## 17.2. 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                             commandBuffer,
    uint32_t                                    attachmentCount,
    const VkClearAttachment*                    pAttachments,
    uint32_t                                    rectCount,
    const VkClearRect*                          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.
- `rectCount` is the number of entries in the `pRects` array.
- `pRects` points 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.

## Valid Usage

- If the `aspectMask` member of any given element of `pAttachments` contains `VK_IMAGE_ASPECT_COLOR_BIT`, the `colorAttachment` member of those elements **must** refer to a valid color attachment in the current subpass
- The rectangular region specified by a given element of `pRects` **must** be contained within the render area of the current render pass instance
- The layers specified by a given element of `pRects` **must** be contained within every attachment that `pAttachments` refers to
- 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 pointer to an array of `attachmentCount` valid `VkClearAttachment` structures
- `pRects` **must** be a 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<br>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. `aspectMask` **can** include `VK_IMAGE_ASPECT_COLOR_BIT` for color attachments, `VK_IMAGE_ASPECT_DEPTH_BIT` for depth/stencil attachments with a depth component, and `VK_IMAGE_ASPECT_STENCIL_BIT` for depth/stencil attachments with a stencil component. If the subpass's depth/stencil attachment is `VK_ATTACHMENT_UNUSED`, then the clear has no effect.
- `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. If `colorAttachment` is `VK_ATTACHMENT_UNUSED` then the clear has no effect.
- `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`
- `clearValue` **must** be a valid `VkClearValue` union

### Valid Usage (Implicit)

- `aspectMask` **must** be a valid combination of VkImageAspectFlagBits values
- `aspectMask` **must** not be `0`

## 17.3. 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**

- `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.

**Valid Usage**

- `depthStencil` **must** be a valid `VkClearDepthStencilValue` structure

## 17.4. 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

### 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<br>Secondary | Outside | Transfer<br>graphics<br>compute | Transfer |

## 17.5. Updating Buffers

To update buffer data inline in a command buffer, call:

```
void vkCmdUpdateBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    dstBuffer,
    VkDeviceSize                                dstOffset,
    VkDeviceSize                                dataSize,
    const void*                                 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.

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

### Valid Usage (Implicit)

- commandBuffer **must** be a valid VkCommandBuffer handle
- dstBuffer **must** be a valid VkBuffer handle
- pData **must** be a 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<br>Secondary | Outside | Transfer<br>graphics<br>compute | Transfer |

*Note*

The `pData` parameter was of type `uint32_t*` instead of `void*` prior to revision 1.0.19 of the Specification and VK_HEADER_VERSION 19 of vulkan.h. This was a historical anomaly, as the source data may be of other types.

# Chapter 18. 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`.

## 18.1. Common Operation

Some rules for valid operation are common to all copy commands:

- Copy commands **must** be recorded outside of a render pass instance.
- For non-sparse resources, the union of the source regions in a given buffer or image **must** not overlap the union of the destination regions in the same buffer or image.
- For sparse resources, the set of bytes used by all the source regions **must** not intersect the set of bytes used by all the destination regions.
- 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** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by vkGetPhysicalDeviceFormatProperties.
- Destination images **must** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by vkGetPhysicalDeviceFormatProperties.
- Source images **must** have been created with the `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage bit enabled and destination images **must** have been created with the `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage bit enabled.
- 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.

All copy commands are treated as “transfer” operations for the purposes of synchronization barriers.

## 18.2. Copying Data Between Buffers

To copy data between buffer objects, call:

```
void vkCmdCopyBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    srcBuffer,
    VkBuffer                                    dstBuffer,
    uint32_t                                    regionCount,
    const VkBufferCopy*                         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 result is undefined if the copy regions overlap in memory.

### Valid Usage

- The `size` member of a given element of `pRegions` **must** be greater than `0`
- The `srcOffset` member of a given element of `pRegions` **must** be less than the size of `srcBuffer`
- The `dstOffset` member of a given element of `pRegions` **must** be less than the size of `dstBuffer`
- The `size` member of a given element of `pRegions` **must** be less than or equal to the size of `srcBuffer` minus `srcOffset`
- The `size` member of a given 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

### 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 pointer to an array of `regionCount` `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<br>Secondary | Outside | Transfer<br>graphics<br>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.

## 18.3. 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                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkImage                                     dstImage,
    VkImageLayout                               dstImageLayout,
    uint32_t                                    regionCount,
    const VkImageCopy*                          pRegions);
```

- 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 considered compatible if their element size is the same between both formats. For example, VK_FORMAT_R8G8B8A8_UNORM is compatible with VK_FORMAT_R32_UINT because both texels are 4 bytes in size. Depth/stencil formats **must** match exactly.

vkCmdCopyImage allows copying between size-compatible compressed and uncompressed internal formats. Formats are size-compatible if the element size of the uncompressed format is equal to the element size (compressed 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 4x4 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 4x4 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.

`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 a given element of `pRegions` **must** be a region that is contained within `srcImage`
- The destination region specified by a given 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
- `srcImage` **must** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by vkGetPhysicalDeviceFormatProperties
- `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`
- `dstImage` **must** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by vkGetPhysicalDeviceFormatProperties
- `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 VkFormat of each of `srcImage` and `dstImage` **must** be compatible, as defined below
- The sample count of `srcImage` and `dstImage` **must** match

## 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 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<br>Secondary | Outside | Transfer<br>graphics<br>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 source 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.

## Valid Usage

- The `aspectMask` member of `srcSubresource` and `dstSubresource` **must** match
- 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 (`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`.
- `srcSubresource.baseArrayLayer` **must** be less than and (`srcSubresource.layerCount` + `srcSubresource.baseArrayLayer`) **must** be less than or equal to the number of layers in the source image
- `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`.
- `dstSubresource.baseArrayLayer` **must** be less than and (`dstSubresource.layerCount` + `dstSubresource.baseArrayLayer`) **must** be less than or equal to the number of layers in the destination image
- If the calling command's `srcImage` is a compressed format image, 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 format image, extent.width **must** be a multiple of the compressed texel block width or (extent.width + srcOffset.x) **must** equal the source image subresource width
- If the calling command's srcImage is a compressed format image, extent.height **must** be a multiple of the compressed texel block height or (extent.height + srcOffset.y) **must** equal the source image subresource height
- If the calling command's srcImage is a compressed format image, extent.depth **must** be a multiple of the compressed texel block depth or (extent.depth + srcOffset.z) **must** equal the source image subresource depth
- If the calling command's dstImage is a compressed format image, 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, extent.width **must** be a multiple of the compressed texel block width or (extent.width + dstOffset.x) **must** equal the destination image subresource width
- If the calling command's dstImage is a compressed format image, extent.height **must** be a multiple of the compressed texel block height or (extent.height + dstOffset.y) **must** equal the destination image subresource height
- If the calling command's dstImage is a compressed format image, extent.depth **must** be a multiple of the compressed texel block depth or (extent.depth + dstOffset.z) **must** equal the destination image subresource depth
- srcOffset, dstOffset, and extent **must** respect the image transfer granularity requirements of the queue family that it will be submitted against, as described in Physical Device Enumeration

### 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`
- `mipLevel` **must** be less than the `mipLevels` specified in VkImageCreateInfo when the image was created
- (`baseArrayLayer` + `layerCount`) **must** be less than or equal to the `arrayLayers` specified in VkImageCreateInfo when the image was created

### Valid Usage (Implicit)

- `aspectMask` **must** be a valid combination of VkImageAspectFlagBits values
- `aspectMask` **must** not be `0`

## 18.4. Copying Data Between Buffers and Images

To copy data from a buffer object to an image object, call:

```
void vkCmdCopyBufferToImage(
    VkCommandBuffer                                 commandBuffer,
    VkBuffer                                        srcBuffer,
    VkImage                                         dstImage,
    VkImageLayout                                   dstImageLayout,
    uint32_t                                        regionCount,
    const VkBufferImageCopy*                        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.

## Valid Usage

- The buffer region specified by a given element of `pRegions` **must** be a region that is contained within `srcBuffer`
- The image region specified by a given 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
- `srcBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag
- `dstImage` **must** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by vkGetPhysicalDeviceFormatProperties
- 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_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

## 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 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<br>Secondary | Outside | Transfer<br>graphics<br>compute | Transfer |

To copy data from an image object to a buffer object, call:

```
void vkCmdCopyImageToBuffer(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkBuffer                                    dstBuffer,
    uint32_t                                    regionCount,
    const VkBufferImageCopy*                    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.

## Valid Usage

- The image region specified by a given element of `pRegions` **must** be a region that is contained within `srcImage`
- The buffer region specified by a given element of `pRegions` **must** be a region that is contained within `dstBuffer`
- The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory
- `srcImage` **must** use a format that supports `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by vkGetPhysicalDeviceFormatProperties
- `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

## 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 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<br>Secondary | Outside | Transfer<br>graphics<br>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 the data in buffer memory as a subregion of a larger two- or three-dimensional image, and control the addressing calculations of data in buffer memory. 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, the data in buffer memory **must** be in the the range [0,1] or undefined results occur.

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 the calling command's `VkImage` parameter's format is not a depth/stencil format, then `bufferOffset` **must** be a multiple of the format's element size
- `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
- `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
- 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 format image, `bufferRowLength` **must** be a multiple of the compressed texel block width
- If the calling command's `VkImage` parameter is a compressed format image, `bufferImageHeight` **must** be a multiple of the compressed texel block height
- If the calling command's `VkImage` parameter is a compressed format image, 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 format image, `bufferOffset` **must** be a multiple of the compressed texel block size in bytes
- If the calling command's `VkImage` parameter is a compressed format image, `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 format image, `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 format image, `imageExtent.depth` **must** be a multiple of the compressed texel block depth or (`imageExtent.depth` + `imageOffset.z`) **must** equal the image subresource depth
- `bufferOffset`, `bufferRowLength`, `bufferImageHeight` and all members of `imageOffset` and `imageExtent` **must** respect the image transfer granularity requirements of the queue family that it will be submitted against, as described in Physical Device Enumeration
- The `aspectMask` member of `imageSubresource` **must** specify aspects present in the calling

command's VkImage parameter

- 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
- When copying to the depth aspect of an image subresource, the data in the source buffer **must** be in the range [0,1]

### Valid Usage (Implicit)

- imageSubresource **must** be a valid VkImageSubresourceLayers structure

Pseudocode for image/buffer addressing is:

```
rowLength = region->bufferRowLength;
if (rowLength == 0)
    rowLength = region->imageExtent.width;

imageHeight = region->bufferImageHeight;
if (imageHeight == 0)
    imageHeight = region->imageExtent.height;

elementSize = <element size of the format of the src/dstImage>;

address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x)
* elementSize;

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-compression 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.

## 18.5. 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                                 commandBuffer,
    VkImage                                         srcImage,
    VkImageLayout                                   srcImageLayout,
    VkImage                                         dstImage,
    VkImageLayout                                   dstImageLayout,
    uint32_t                                        regionCount,
    const VkImageBlit*                              pRegions,
    VkFilter                                        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:

  $u_{base} = i + ½$

  $v_{base} = j + ½$

  $w_{base} = k + ½$

- These base coordinates are then offset by the first destination offset:

  $u_{offset} = u_{base} - x_{dst0}$

  $v_{offset} = v_{base} - y_{dst0}$

  $w_{offset} = w_{base} - z_{dst0}$

  $a_{offset} = a - \texttt{baseArrayCount}_{dst}$

- The scale is determined from the source and destination regions, and applied to the offset coordinates:

  $scale_u = (x_{src1} - x_{src0}) / (x_{dst1} - x_{dst0})$

  $scale_v = (y_{src1} - y_{src0}) / (y_{dst1} - y_{dst0})$

  $scale_w = (z_{src1} - z_{src0}) / (z_{dst1} - z_{dst0})$

  $u_{scaled} = u_{offset} * scale_u$

  $v_{scaled} = v_{offset} * scale_v$

  $w_{scaled} = w_{offset} * scale_w$

- Finally the source offset is added to the scaled coordinates, to determine the final unnormalized coordinates used to sample from `srcImage`:

  $u = u_{scaled} + x_{src0}$

  $v = v_{scaled} + y_{src0}$

  $w = w_{scaled} + z_{src0}$

  $q =$ `mipLevel`

  $a = a_{offset} +$ `baseArrayCount`$_{src}$

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 value sampled from the source image is taken from the single nearest slice (with undefined 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 a given element of `pRegions` **must** be a region that is contained within `srcImage`
- The destination region specified by a given 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
- `srcImage` **must** use a format that supports `VK_FORMAT_FEATURE_BLIT_SRC_BIT`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by `vkGetPhysicalDeviceFormatProperties`
- `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`
- `dstImage` **must** use a format that supports `VK_FORMAT_FEATURE_BLIT_DST_BIT`, which is indicated by `VkFormatProperties::linearTilingFeatures` (for linearly tiled images) or `VkFormatProperties::optimalTilingFeatures` (for optimally tiled images) - as returned by `vkGetPhysicalDeviceFormatProperties`
- `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`, `srcImage` **must** be of a format which supports linear filtering, as specified by the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` flag in `VkFormatProperties::linearTilingFeatures` (for a linear image) or `VkFormatProperties::optimalTilingFeatures`(for an optimally tiled image) returned by `vkGetPhysicalDeviceFormatProperties`
- If `filter` is `VK_FILTER_CUBIC_IMG`, `srcImage` **must** be of a format which supports cubic filtering, as specified by the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG` flag in `VkFormatProperties::linearTilingFeatures` (for a linear image) or `VkFormatProperties::optimalTilingFeatures`(for an optimally tiled image) returned by `vkGetPhysicalDeviceFormatProperties`
- If `filter` is `VK_FILTER_CUBIC_IMG`, `srcImage` **must** have a VkImageType of `VK_IMAGE_TYPE_3D`

## 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 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<br>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 an array of two VkOffset3D structures specifying the bounds of the source region within `srcSubresource`.
- `dstSubresource` is the subresource to blit into.
- `dstOffsets` is 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

## 18.6. Resolving Multisample Images

To resolve a multisample image to a non-multisample image, call:

```
void vkCmdResolveImage(
    VkCommandBuffer                                 commandBuffer,
    VkImage                                         srcImage,
    VkImageLayout                                   srcImageLayout,
    VkImage                                         dstImage,
    VkImageLayout                                   dstImageLayout,
    uint32_t                                        regionCount,
    const VkImageResolve*                           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 a given element of `pRegions` **must** be a region that is contained within `srcImage`
- The destination region specified by a given 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`
- If `dstImage` was created with `tiling` equal to `VK_IMAGE_TILING_LINEAR`, `dstImage` **must** have been created with a `format` that supports being a color attachment, as specified by the `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` flag in `VkFormatProperties::linearTilingFeatures` returned by `vkGetPhysicalDeviceFormatProperties`
- If `dstImage` was created with `tiling` equal to `VK_IMAGE_TILING_OPTIMAL`, `dstImage` **must** have been created with a `format` that supports being a color attachment, as specified by the `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` flag in `VkFormatProperties::optimalTilingFeatures` returned by `vkGetPhysicalDeviceFormatProperties`

## 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 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<br>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

# Chapter 19. 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 currently bound graphics pipeline. A graphics pipeline **must** be bound to a command buffer before any drawing commands are recorded in that command buffer.

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`, 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

## 19.1. 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. Supported topologies are defined by VkPrimitiveTopology and include:

```
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,
} VkPrimitiveTopology;
```

Each primitive topology, and its construction from a list of vertices, is summarized below.

> *Note*
>
> The terminology “the vertex i ” means “the vertex with index i in the ordered list of vertices defining this primitive”.

### 19.1.1. Points

A series of individual points are specified with `topology` `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`. Each vertex defines a separate point.

### 19.1.2. Separate Lines

Individual line segments, each defined by a pair of vertices, are specified with `topology` `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`. The first two vertices define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of vertices is odd, then the last vertex is ignored.

### 19.1.3. Line Strips

A series of one or more connected line segments are specified with `topology` `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`. In this case, the first vertex specifies the first segment’s start point while the second vertex specifies the first segment’s endpoint and the second segment’s start point. In general, vertex i (for i > 0) specifies the beginning of the ith segment and the end of the previous segment. The last vertex specifies the end of the last segment. If only one vertex is specified, then no primitive is generated.

### 19.1.4. Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and is specified with `topology` `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`. In this case, the first three vertices define the first triangle, and their order is significant. Each subsequent vertex defines a new triangle using that point along with the last two vertices from the previous triangle, as shown in figure Triangle strips, fans, and lists. If fewer than three vertices are specified, no primitive is produced. The order of vertices in successive triangles changes as shown in the figure, so that all triangle faces have the same orientation.

*Figure 5. Triangle strips, fans, and lists*

**Caption**

In the Triangle strips, fans, and lists diagram, the sub-sections represent:

- (a) A triangle strip.
- (b) A triangle fan.
- (c) Independent triangles.

The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

### 19.1.5. Triangle Fans

A triangle fan is specified with `topology` `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`. It is similar to a triangle strip, but changes the vertex replaced from the previous triangle as shown in figure Triangle strips, fans, and lists, so that all triangles in the fan share a common vertex.

### 19.1.6. Separate Triangles

Separate triangles are specified with `topology` `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, as shown in figure Triangle strips, fans, and lists. In this case, vertices 3 i, 3 i + 1, and 3 i + 2 (in that order) determine a triangle for each i = 0, 1, ..., n-1, where there are 3 n + k vertices drawn. k is either 0, 1, or 2; if k is not zero, the final k vertices are ignored.

### 19.1.7. Lines With Adjacency

Lines with adjacency are specified with `topology` `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, and are independent line segments where each endpoint has a corresponding *adjacent* vertex that is accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

A line segment is drawn from vertex 4 i + 1 to vertex 4 i + 2 for each i = 0, 1, ..., n-1, where there are 4 n + k vertices. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line

segment i, vertices 4 i and 4 i + 3 vertices are considered adjacent to vertices 4 i + 1 and 4 i + 2, respectively, as shown in figure Lines with adjacency.

*Figure 6. Lines with adjacency*

**Caption**

In the Lines with adjacency diagram, the sub-sections represent:

- (a) Lines with adjacency.
- (b) Line strips with adjacency.

The vertices connected with solid lines belong to the main primitives. The vertices connected by dashed lines are the adjacent vertices that are accessible in a geometry shader.

### 19.1.8. Line Strips With Adjacency

Line strips with adjacency are specified with `topology` `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY` and are similar to line strips, except that each line segment has a pair of adjacent vertices that are accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

A line segment is drawn from vertex i + 1 vertex to vertex i + 2 for each i = 0, 1, ..., n-1, where there are n + 3 vertices. If there are fewer than four vertices, all vertices are ignored. For line segment i, vertices i and i + 3 are considered adjacent to vertices i + 1 and i + 2, respectively, as shown in figure Lines with adjacency.

### 19.1.9. Triangle List With Adjacency

Triangles with adjacency are specified with `topology`

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY, and are similar to separate triangles except that each triangle edge has an adjacent vertex that is accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

Vertices 6 i, 6 i + 2, and 6 i + 4 (in that order) determine a triangle for each i = 0, 1, ..., n-1, where there are 6 n+k vertices. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, vertices 6 i + 1, 6 i + 3, and 6 i + 5 vertices are considered adjacent to edges from vertex 6 i to 6 i + 2, from 6 i + 2 to 6 i + 4, and from 6 i + 4 to 6 i vertices, respectively, as shown in figure Triangles with adjacency.

*Figure 7. Triangles with adjacency*

**Caption**

In the Triangles with adjacency diagram, the vertices connected with solid lines belong to the main primitive. The vertices connected by dashed lines are the adjacent vertices that are accessible in a geometry shader.

### 19.1.10. Triangle Strips With Adjacency

Triangle strips with adjacency are specified with topology VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY, and are similar to triangle strips except that each triangle edge has an adjacent vertex that is accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

*Figure 8. Triangle strips with adjacency*

## Caption

In the Triangle strips with adjacency diagram, the vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that are accessible in a geometry shader.

In triangle strips with adjacency, n triangles are drawn where there are 2 (n + 2) + k vertices. k is either 0 or 1; if k is 1, the final vertex is ignored. If there are fewer than 6 vertices, the entire primitive is ignored. Table Triangles generated by triangle strips with adjacency. describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle, as shown in figure Triangle strips with adjacency.

*Table 22. Triangles generated by triangle strips with adjacency.*

| | **Primitive Vertices** | | | **Adjacent Vertices** | | |
|---|---|---|---|---|---|---|
| Primitive | 1st | 2nd | 3rd | 1/2 | 2/3 | 3/1 |
| only (i = 0, n = 1) | 0 | 2 | 4 | 1 | 5 | 3 |
| first (i = 0) | 0 | 2 | 4 | 1 | 6 | 3 |
| middle (i odd) | 2 i + 2 | 2 i | 2 i + 4 | 2 i-2 | 2 i + 3 | 2 i + 6 |

| | Primitive Vertices | | | Adjacent Vertices | | |
|---|---|---|---|---|---|---|
| middle (i even) | 2 i | 2 i + 2 | 2 i + 4 | 2 i-2 | 2 i + 6 | 2 i + 3 |
| last (i=n-1, i odd) | 2 i + 2 | 2 i | 2 i + 4 | 2 i-2 | 2 i + 3 | 2 i + 5 |
| last (i=n-1, i even) | 2 i | 2 i + 2 | 2 i + 4 | 2 i-2 | 2 i + 5 | 2 i + 3 |

**Caption**

In the Triangles generated by triangle strips with adjacency table, each triangle is drawn using the vertices whose numbers are in the **1st**, **2nd**, and **3rd** columns under **Primitive Vertices**, in that order. The vertices in the 1/2, 2/3, and 3/1 columns under **Adjacent Vertices** are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to six cases: the first and only triangle (i = 0, n = 1), the first triangle of several (i = 0, n > 0), *odd* middle triangles (i = 1, 3, 5 ...), *even* middle triangles (i = 2, 4, 6, ...), and special cases for the last triangle, when i is either even or odd. For the purposes of this table, both the first vertex and first triangle are numbered 0.

### 19.1.11. Separate Patches

Separate patches are specified with `topology` `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`. A patch is an ordered collection of vertices used for primitive tessellation. The vertices comprising a patch have no implied geometric ordering, and are used by tessellation shaders and the fixed-function tessellator to generate new point, line, or triangle primitives.

Each patch in the series has a fixed number of vertices, specified by the `patchControlPoints` member of the VkPipelineTessellationStateCreateInfo structure passed to vkCreateGraphicsPipelines. Once assembled and vertex shaded, these patches are provided as input to the tessellation control shader stage.

If the number of vertices in a patch is given by v, vertices v × i through v × i + v - 1 (in that order) determine a patch for each i = 0, 1, ..., n-1, where there are v × n + k vertices. k is in the range [0, v - 1]; if k is not zero, the final k vertices are ignored.

### 19.1.12. General Considerations For Polygon Primitives

Depending on the polygon mode, a *polygon primitive* generated from a drawing command with `topology` `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY`, or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY` is rendered in one of several ways, such as outlining its border or filling its interior. The order of vertices in such a primitive is significant during polygon rasterization and fragment shading.

## 19.2. 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.

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`, 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.

## 19.3. 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 undefined 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
    - 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
  - 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` selects whether indices are treated as 16 bits or 32 bits. Possible values include:

```
typedef enum VkIndexType {
    VK_INDEX_TYPE_UINT16 = 0,
    VK_INDEX_TYPE_UINT32 = 1,
} VkIndexType;
```

### 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

### 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<br>Secondary | Both | Graphics | |

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                                 commandBuffer,
    uint32_t                                        vertexCount,
    uint32_t                                        instanceCount,
    uint32_t                                        firstVertex,
    uint32_t                                        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 currently bound graphics pipeline.

## Valid Usage

- The current render pass **must** be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` currently 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- 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
- A valid graphics pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_GRAPHICS`
- If the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` requires any dynamic state, that state **must** have been set on the current command buffer
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- If any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` uses unnormalized coordinates, it **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 any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` uses unnormalized coordinates, it **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 any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` uses unnormalized coordinates, it **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 any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** be of a format which supports cubic filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** not have a VkImageViewType of VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to VkPhysicalDeviceMultiviewPropertiesKHX::maxMultiviewInstanceIndex.

## 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<br>Secondary | Inside | Graphics | Graphics |

To record an indexed draw, call:

```
void vkCmdDrawIndexed(
    VkCommandBuffer                                 commandBuffer,
    uint32_t                                        indexCount,
    uint32_t                                        instanceCount,
    uint32_t                                        firstIndex,
    int32_t                                         vertexOffset,
    uint32_t                                        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 currently 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_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 currently bound graphics pipeline.

## Valid Usage

- The current render pass **must** be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` currently 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- 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
- A valid graphics pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_GRAPHICS`
- If the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` requires any dynamic state, that state **must** have been set on the current command buffer
- (`indexSize` * (`firstIndex` + `indexCount`) + `offset`) **must** be less than or equal to the size of the currently bound index buffer, with indexSize being based on the type specified by `indexType`, where the index buffer, `indexType`, and `offset` are specified via `vkCmdBindIndexBuffer`
- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- If any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` uses unnormalized coordinates, it **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 any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** be of a format which supports cubic filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** not have a VkImageViewType of VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to VkPhysicalDeviceMultiviewPropertiesKHX::maxMultiviewInstanceIndex.

## 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<br>Secondary | Inside | Graphics | Graphics |

To record a non-indexed indirect draw, call:

```
void vkCmdDrawIndirect(
    VkCommandBuffer                                 commandBuffer,
    VkBuffer                                        buffer,
    VkDeviceSize                                    offset,
    uint32_t                                        drawCount,
    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.
- `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 `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `offset` **must** be a multiple of `4`
- If `drawCount` is greater than `1`, `stride` **must** be a multiple of `4` and **must** be greater than or equal to sizeof(`VkDrawIndirectCommand`)
- If the multi-draw indirect feature is not enabled, `drawCount` **must** be `0` or `1`
- If the drawIndirectFirstInstance feature is not enabled, all the `firstInstance` members of the `VkDrawIndirectCommand` structures accessed by this command **must** be `0`
- The current render pass **must** be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` currently 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- A valid graphics pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_GRAPHICS`
- If the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` requires any dynamic state, that state **must** have been set on the current command buffer
- 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`
- `drawCount` **must** be less than or equal to `VkPhysicalDeviceLimits`::`maxDrawIndirectCount`
- Every input attachment used by the current subpass **must** be bound to the pipeline via a

descriptor set

- If any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** be of a format which supports cubic filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** not have a VkImageViewType of VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to VkPhysicalDeviceMultiviewPropertiesKHX::maxMultiviewInstanceIndex.

## 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<br>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 vkCmdDrawIndirectCountAMD(
    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.

`vkCmdDrawIndirectCountAMD` behaves similar 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

- `offset` **must** be a multiple of `4`
- `countBufferOffset` **must** be a multiple of `4`
- `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`, (`stride` × (`maxDrawCount` - 1) + `offset` + sizeof(`VkDrawIndirectCommand`)) **must** be less than or equal to the size of `buffer`
- If the drawIndirectFirstInstance feature is not enabled, all the `firstInstance` members of the `VkDrawIndirectCommand` structures accessed by this command **must** be `0`
- The current render pass **must** be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` currently 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- A valid graphics pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_GRAPHICS`
- If the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` requires any dynamic state, that state **must** have been set on the current command buffer
- If the count stored in `countBuffer` is equal to `1`, (`offset` + sizeof(`VkDrawIndirectCommand`)) **must** be less than or equal to the size of `buffer`
- If the count stored in `countBuffer` is greater than `1`, (`stride` × (`drawCount` - 1) + `offset` sizeof(`VkDrawIndirectCommand`)) **must** be less than or equal to the size of `buffer`
- The count stored in `countBuffer` **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`

- Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set
- If any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to VkPhysicalDeviceMultiviewPropertiesKHX::maxMultiviewInstanceIndex.

## 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<br>Secondary | Inside | Graphics | Graphics |

To record an indexed indirect draw, call:

```
void vkCmdDrawIndexedIndirect(
    VkCommandBuffer                                 commandBuffer,
    VkBuffer                                        buffer,
    VkDeviceSize                                    offset,
    uint32_t                                        drawCount,
    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.
- `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 `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- `offset` **must** be a multiple of `4`
- 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 multi-draw indirect feature is not enabled, `drawCount` **must** be `0` or `1`
- If the drawIndirectFirstInstance feature is not enabled, all the `firstInstance` members of the `VkDrawIndexedIndirectCommand` structures accessed by this command **must** be `0`
- The current render pass **must** be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` currently 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- A valid graphics pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_GRAPHICS`
- If the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` requires any dynamic state, that state **must** have been set on the current command buffer
- 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`
- `drawCount` **must** be less than or equal to `VkPhysicalDeviceLimits`::`maxDrawIndirectCount`
- Every input attachment used by the current subpass **must** be bound to the pipeline via a

descriptor set

- If any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** be of a format which supports cubic filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** not have a VkImageViewType of VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to VkPhysicalDeviceMultiviewPropertiesKHX::maxMultiviewInstanceIndex.

## 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<br>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 currently 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 vkCmdDrawIndexedIndirectCountAMD(
    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.

vkCmdDrawIndexedIndirectCountAMD behaves similar to vkCmdDrawIndirectCountAMD 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

- `offset` **must** be a multiple of `4`
- `countBufferOffset` **must** be a multiple of `4`
- `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`, (`stride` × (`maxDrawCount` - 1) + `offset` + sizeof(`VkDrawIndirectCommand`)) **must** be less than or equal to the size of `buffer`
- If the drawIndirectFirstInstance feature is not enabled, all the `firstInstance` members of the `VkDrawIndexedIndirectCommand` structures accessed by this command **must** be `0`
- The current render pass **must** be compatible with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` currently 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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` currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_GRAPHICS`, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have valid buffers bound
- A valid graphics pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_GRAPHICS`
- If the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` requires any dynamic state, that state **must** have been set on the current command buffer
- If count stored in `countBuffer` is equal to `1`, (`offset` + sizeof(`VkDrawIndexedIndirectCommand`)) **must** be less than or equal to the size of `buffer`
- If count stored in `countBuffer` is greater than `1`, (`stride` × (`drawCount` - 1) + `offset` sizeof(`VkDrawIndexedIndirectCommand`)) **must** be less than or equal to the size of `buffer`
- `drawCount` **must** be less than or equal to `VkPhysicalDeviceLimits`::`maxDrawIndirectCount`
- Every input attachment used by the current subpass **must** be bound to the pipeline via a

descriptor set

- If any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any VkSampler object that is accessed from a shader by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it **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 any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to VkPhysicalDeviceMultiviewPropertiesKHX::maxMultiviewInstanceIndex.

## 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<br>Secondary | Inside | Graphics | Graphics |

# Chapter 20. Fixed-Function Vertex Processing

Some implementations have specialized fixed-function hardware for fetching and format-converting vertex input data from buffers, rather than performing the fetch as part of the vertex shader. Vulkan includes a vertex attribute fetch stage in the graphics pipeline in order to take advantage of this.

## 20.1. 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.

*SPIR-V example*

```
        ...
 %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
        ...
```

### 20.1.1. 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 23. Input attribute components accessed by 32-bit input variables*

| 32-bit data type | Component decoration | Components consumed |
|---|---|---|
| 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 24. Input attributes accessed by 32-bit input matrix variables*

| Data type | Column vector type | Locations consumed | Components consumed |
|---|---|---|---|
| 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 25. 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.

## 20.2. Vertex Input Description

Applications specify vertex input attribute and vertex input binding descriptions as part of graphics pipeline creation. The VkGraphicsPipelineCreateInfo::pVertexInputState points to a structure of type VkPipelineVertexInputStateCreateInfo.

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 any given 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
- flags **must** be 0
- If vertexBindingDescriptionCount is not 0, pVertexBindingDescriptions **must** be a pointer to an array of vertexBindingDescriptionCount valid VkVertexInputBindingDescription structures
- If vertexAttributeDescriptionCount is not 0, pVertexAttributeDescriptions **must** be a pointer to an array of vertexAttributeDescriptionCount valid VkVertexInputAttributeDescription structures

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 specifies whether vertex attribute addressing is a function of the vertex index or of the instance index. Possible values include:

  ```
  typedef enum VkVertexInputRate {
      VK_VERTEX_INPUT_RATE_VERTEX = 0,
      VK_VERTEX_INPUT_RATE_INSTANCE = 1,
  } VkVertexInputRate;
  ```

  - VK_VERTEX_INPUT_RATE_VERTEX indicates that vertex attribute addressing is a function of the vertex index.
  - VK_VERTEX_INPUT_RATE_INSTANCE indicates that vertex attribute addressing is a function 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

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                             commandBuffer,
    uint32_t                                    firstBinding,
    uint32_t                                    bindingCount,
    const VkBuffer*                             pBuffers,
    const VkDeviceSize*                         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 pointer to an array of `bindingCount` valid `VkBuffer` handles
- `pOffsets` **must** be a 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<br>Secondary | Both | Graphics | |

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`).

```
bufferBindingAddress = buffer[binding].baseAddress + offset[binding];

if (bindingDesc.inputRate == VK_VERTEX_INPUT_RATE_VERTEX)
    vertexOffset = vertexIndex * bindingDesc.stride;
else
    vertexOffset = instanceIndex * 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.

## 20.3. 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 21. 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.

## 21.1. 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 figure Domain parameterization for tessellation primitive modes.

*Figure 9. Domain parameterization for tessellation primitive modes*

## Caption

In the Domain parameterization diagram, 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.

## 21.2. 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.

## 21.3. 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, `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, `maxLevel`] and then rounded up to the nearest even integer n. If `SpacingFractionalOdd` is used, the tessellation level is clamped to [1, `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 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.

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 positive, and clockwise ordering if a is negative. $u_i$ and $v_i$ are the u and v coordinates in

normalized parameter space of the ith vertex of the triangle.

> *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.

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. Otherwise, the tessellator will produce a collection of line segments or triangles according to the primitive mode. When tessellating triangles or quads in 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.

## 21.4. 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.

## 21.5. 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 + ε 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 10. 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.

The order in which the vertices for a given output triangle is generated is implementation-dependent. However, when depicted in a manner similar to Inner Triangle Tessellation, the order of the vertices in each generated triangle will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

## 21.6. 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 + ε 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 11. Inner Quad Tessellation*

## Caption

In the Inner Quad Tessellation diagram, inner quad tessellation levels of (a) (4,2) and (b) (7,4) are shown. Gray regions 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 triangle. 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.

The order in which the vertices for a given output triangle is generated is implementation-dependent. However, when depicted in a manner similar to Inner Quad Tessellation, the order of the vertices in each generated triangle will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

## 21.7. 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, $\frac{1}{n}$, $\frac{2}{n}$, …, $\frac{n-1}{n}$.

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.

The order in which the vertices for a given output line is generated is implementation-dependent.

## 21.8. Tessellation Pipeline State

The `pTessellationState` member of VkGraphicsPipelineCreateInfo points to a structure of type `VkPipelineTessellationStateCreateInfo`.

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`
- `flags` **must** be `0`

# Chapter 22. 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.

## 22.1. 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** be compatible with the primitive topology used by primitive assembly (if tessellation is not in use) or **must** match the type of primitive generated by the tessellation primitive generator (if tessellation is in use). Compatibility is defined below, with each input primitive type. The input primitive types accepted by a geometry shader are:

**Points**

Geometry shaders that operate on points use an `OpExecutionMode` instruction specifying the `InputPoints` input mode. Such a shader is valid only when the pipeline primitive topology is `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` (if tessellation is not in use) or if tessellation is in use and the tessellation evaluation shader uses `PointMode`. There is only a single input vertex available for each geometry shader invocation. However, inputs to the geometry shader are still presented as an array, but this array has a length of one.

**Lines**

Geometry shaders that operate on line segments are generated by including an `OpExecutionMode` instruction with the `InputLines` mode. Such a shader is valid only for the `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`, and `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP` primitive topologies (if tessellation is not in use) or if tessellation is in use and the tessellation mode is `Isolines`. There are two input vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment.

**Lines with Adjacency**

Geometry shaders that operate on line segments with adjacent vertices are generated by including an `OpExecutionMode` instruction with the `InputLinesAdjacency` mode. Such a shader is valid only for the `VK_PRIMITIVE_TOPOLOGY_LINES_WITH_ADJACENCY` and `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY` primitive topologies and **must** not be used when tessellation is in use.

In this mode, there are four vertices available for each geometry shader invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

**Triangles**

Geometry shaders that operate on triangles are created by including an `OpExecutionMode` instruction with the `Triangles` mode. Such a shader is valid when the pipeline topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN` (if tessellation is not in use) or when tessellation is in use and the tessellation mode is `Triangles` or `Quads`.

In this mode, there are three vertices available for each geometry shader invocation. The first, second, and third vertices refer to attributes of the first, second, and third vertex of the triangle, respectively.

**Triangles with Adjacency**

Geometry shaders that operate on triangles with adjacent vertices are created by including an `OpExecutionMode` instruction with the `InputTrianglesAdjacency` mode. Such a shader is valid when the pipeline topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLES_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`, and **must** not be used when tessellation is in use.

In this mode, there are six vertices available for each geometry shader invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

## 22.2. 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.

## 22.3. 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 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`.

## 22.4. 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.

## 22.5. 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.

### 22.5.1. `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.

### 22.5.2. 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 23. Fixed-Function Vertex Post-Processing

After programmable vertex processing, the following fixed-function operations are applied to vertices of the resulting primitives:

- Viewport swizzle (see Viewport Swizzle)
- Flatshading (see Flatshading).
- 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.

## 23.1. 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`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `viewportCount` **must** be greater than `0`

The `VkPipelineViewportSwizzleStateCreateInfoNV` state is set by adding an instance of this structure to the `pNext` chain of an instance of the `VkPipelineViewportStateCreateInfo` 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 the swizzle operation to apply to the x component of the primitive
- `y` is the swizzle operation to apply to the y component of the primitive
- `z` is the swizzle operation to apply to the z component of the primitive
- `w` is 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

The `VkViewportCoordinateSwizzleNV` enum is defined as:

```
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,
} VkViewportCoordinateSwizzleNV;
```

## 23.2. 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. The provoking vertex depends on the primitive topology, and is generally the “first” vertex of the primitive. For primitives not processed by tessellation or geometry shaders, the provoking vertex is selected from the input vertices according to the following table.

*Table 26. Provoking vertex selection*

| Primitive type of primitive i | Provoking vertex number |
|---|---|
| VK_PRIMITIVE_TOPOLOGY_POINT_LIST | i |
| VK_PRIMITIVE_TOPOLOGY_LINE_LIST | 2 i |
| VK_PRIMITIVE_TOPOLOGY_LINE_STRIP | i |
| VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST | 3 i |
| VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP | i |
| VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN | i + 1 |
| VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY | 4 i + 1 |
| VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY | i + 1 |
| VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY | 6 i |
| VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY | 2 i |

**Caption**

The Provoking vertex selection table defines the output values used for flat shading the $i^{th}$ primitive generated by drawing commands with the indicated primitive type, derived from the corresponding values of the vertex whose index is shown in the table. Primitives and vertices are numbered starting from zero.

Flat shading is applied to those vertex attributes that match fragment input attributes which are decorated as `Flat`.

If a geometry shader is active, the output primitive topology is either points, line strips, or triangle strips, and the selection of the provoking vertex behaves according to the corresponding row of the table. If a tessellation evaluation shader is active and a geometry shader is not active, the provoking vertex is undefined but **must** be one of the vertices of the primitive.

## 23.3. 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:

$$-w_c \leq x_c \leq w_c$$
$$-w_c \leq y_c \leq w_c$$
$$0 \leq z_c \leq w_c$$

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 **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.

Depth clamping is enabled or disabled via the `depthClampEnable` enable of the `VkPipelineRasterizationStateCreateInfo` structure. If depth clamping is enabled, 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.

If a point's vertex lies outside of the clip volume, the entire primitive **may** be discarded.

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 **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.

## 23.4. 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 $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (see Primitive Clipping) for a clipped point **P** is used to obtain the output value associated with **P** as

$$\mathbf{c} = t\,\mathbf{c}_1 + (1-t)\,\mathbf{c}_2.$$

(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 $\mathbf{c}_1$ or $\mathbf{c}_2$, since flat shading has already occurred and the two values are identical.

## 23.5. 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.

$$w_c' = x_{coeff}\, x_c + y_{coeff}\, y_c + w_c$$

The VkPipelineViewportWScalingStateCreateInfoNV structure is defined as:

```
typedef struct VkPipelineViewportWScalingStateCreateInfoNV {
    VkStructureType                 sType;
    const void*                     pNext;
    VkBool32                        viewportWScalingEnable;
    uint32_t                        viewportCount;
    const VkViewportWScalingNV*     pViewportWScalings;
} VkPipelineViewportWScalingStateCreateInfoNV;
```

- sType is the type of this structure
- pNext is NULL or a pointer to an extension-specific structure
- viewportWScalingEnable is the enable for viewport W scaling
- 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 VkViewportWScalingNV structures which define the W scaling parameters for the corresponding viewport. If the viewport W scaling state is dynamic, this member is ignored.
- If no element of the pDynamicStates member of sname::VkPipelineViewportStateCreateInfo::pDynamicState is VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV, the pViewportScalings member of a VkPipelineViewportWScalingStateCreateInfoNV extension structure **must** be a pointer to an array of VkPipelineViewportWScalingStateCreateInfoNV::viewportCount VkViewportWScalingNV structures

### Valid Usage

- If the multiple viewports feature is not enabled and viewportWScalingEnable is VK_TRUE, viewportCount **must** be 1
- viewportCount **must** be between 1 and VkPhysicalDeviceLimits::maxViewports, inclusive if viewportWScalingEnable is VK_TRUE
- viewportCount and VkPipelineViewportStateCreateInfo::viewportCount **must** be identical if viewportWScalingEnable is VK_TRUE

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV
- viewportCount **must** be greater than 0

The VkPipelineViewportWScalingStateCreateInfoNV state is set by adding an instance of this structure to the pNext chain of an instance of the VkPipelineViewportStateCreateInfo 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 VkPipelineViewportWScalingStateCreateInfoNV 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 vkCmdSetViewportWScalingNV(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstViewport,
    uint32_t                                    viewportCount,
    const VkViewportWScalingNV*                 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.
- pViewportScalings is a pointer to an array of VkViewportWScalingNV structures specifying viewport parameters.

The viewport parameters taken from element i of pViewportScalings replace the current state for the viewport index firstViewport + i, for i in [0, viewportCount).

### Valid Usage

- The currently 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
- pViewportScalings **must** be a pointer to an array of viewportCount valid VkViewportWScalingNV structures

### 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
- 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<br>Secondary | Both | Graphics | |

Both VkPipelineViewportWScalingStateCreateInfoNV and vkCmdSetViewportWScalingNV use `VkViewportWScalingNV` to set the viewport transformation parameters.

The `VkViewportWScalingNV` 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.

## 23.6. 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

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

## 23.7. 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 which define 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

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkPipelineViewportWScalingStateCreateInfoNV
- `flags` **must** be `0`
- `viewportCount` **must** be greater than `0`
- `scissorCount` **must** be greater than `0`

A *vertex processing stage* may 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 results of 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 `ViewportMask`, 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                                 commandBuffer,
    uint32_t                                        firstViewport,
    uint32_t                                        viewportCount,
    const VkViewport*                               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 currently 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`
- `pViewports` **must** be a pointer to an array of `viewportCount` valid `VkViewport` structures

## 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
- `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<br>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 ∈ { 0, 1, ..., $2^m$-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` + `width` / 2

$o_y$ = `y` + `height` / 2

$o_z$ = `minDepth`

$p_x$ = `width`

$p_y$ = `height`

$p_z$ = `maxDepth` - `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` and less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions`[0]
- `height` **must** be greater than or equal to -`VkPhysicalDeviceLimits::maxViewportDimensions`[1] and less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions`[1]
- If the VK_AMD_negative_viewport_height extension is enabled, `height` **can** also be negative.
- `x` and `y` **must** each be between `viewportBoundsRange`[0] and `viewportBoundsRange`[1], inclusive
- `x` + `width` **must** be less than or equal to `viewportBoundsRange`[1]
- `y` + `height` **must** be less than or equal to `viewportBoundsRange`[1]
- `minDepth` **must** be between `0.0` and `1.0`, inclusive
- `maxDepth` **must** be between `0.0` and `1.0`, inclusive

# Chapter 24. 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 subpixel 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;
    VkPipelineRasterizationStateCreateFlags     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 instead of clipping primitives to the z planes of the frustum, as described in Primitive Clipping.
- `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 the front-facing triangle orientation to be used for culling. See VkFrontFace.
- `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`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkPipelineRasterizationStateRasterizationOrderAMD
- `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

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 per pixel used in rasterization.
- `sampleShadingEnable` specifies that fragment shading executes per-sample if `VK_TRUE`, or per-fragment if `VK_FALSE`, as described in Sample Shading.
- `minSampleShading` is the minimum fraction of sample shading, as described in Sample Shading.
- `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]

**Valid Usage (Implicit)**

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- `rasterizationSamples` **must** be a valid VkSampleCountFlagBits value
- If `pSampleMask` is not `NULL`, `pSampleMask` **must** be a pointer to an array of $\lceil \frac{rasterizationSamples}{32} \rceil$ `VkSampleMask` values

Rasterization only produces fragments corresponding to pixels in the framebuffer. 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.

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. Otherwise, `rasterizationSamples` **must** follow the rules for a zero-attachment subpass.

## 24.1. 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.

## 24.2. 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. Sample mask generation)
3. Depth bounds test
4. Stencil test, stencil op and stencil write
5. Depth test and depth write
6. Sample counting for occlusion queries
7. 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 application **can** select a graphics pipeline to use one of the following primitive rasterization ordering rules:

```
typedef enum VkRasterizationOrderAMD {
    VK_RASTERIZATION_ORDER_STRICT_AMD = 0,
    VK_RASTERIZATION_ORDER_RELAXED_AMD = 1,
} VkRasterizationOrderAMD;
```

- `VK_RASTERIZATION_ORDER_STRICT_AMD` indicates that the order of these operations for each primitive in a subpass **must** occur in primitive order.
- `VK_RASTERIZATION_ORDER_RELAXED_AMD` indicates that the order of these operations for each primitive in a subpass **may** not occur in primitive order.

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 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`.

## 24.3. 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 pixel.

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 pixel including the pixel center or any of the sample locations. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the pixel center **must** be used. The different associated data values need not all be evaluated at the same location. Each pixel 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 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 ith 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 pixel.

*Table 27. 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) | (0.25,0.25)<br>(0.75,0.75) | (0.375, 0.125)<br>(0.875, 0.375)<br>(0.125, 0.625)<br>(0.625, 0.875) | (0.5625, 0.3125)<br>(0.4375, 0.6875)<br>(0.8125, 0.5625)<br>(0.3125, 0.1875)<br>(0.1875, 0.8125)<br>(0.0625, 0.4375)<br>(0.6875, 0.9375)<br>(0.9375, 0.0625) | (0.5625, 0.5625)<br>(0.4375, 0.3125)<br>(0.3125, 0.625)<br>(0.75, 0.4375)<br>(0.1875, 0.375)<br>(0.625, 0.8125)<br>(0.8125, 0.6875)<br>(0.6875, 0.1875)<br>(0.375, 0.875)<br>(0.5, 0.0625)<br>(0.25, 0.125)<br>(0.125, 0.75)<br>(0.0, 0.5)<br>(0.9375, 0.25)<br>(0.875, 0.9375)<br>(0.0625, 0.0) |

## 24.4. Sample Shading

Sample shading **can** be used to specify a minimum number of unique samples to process for each fragment. Sample shading is controlled by the `sampleShadingEnable` member of VkPipelineMultisampleStateCreateInfo. If `sampleShadingEnable` is `VK_FALSE`, sample shading is considered disabled and has no effect. Otherwise, an implementation **must** provide a minimum of max( `minSampleShading` × `rasterizationSamples` , 1) unique associated data for each fragment, where `minSampleShading` is the minimum fraction of sample shading and `rasterizationSamples` is the number of samples requested in VkPipelineMultisampleStateCreateInfo. These are associated with the samples in an implementation-dependent manner. When the sample shading fraction 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.

## 24.5. 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 tessellation control shader, if active and no geometry or tessellation evaluation shader is active; or
- the vertex shader, otherwise

and clamped to the implementation-dependent point size range [`pointSizeRange`[0], `pointSizeRange`[1]]. If the value written to `PointSize` is less than or equal to zero, or if no value was

written to PointSize, results are undefined.

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.

### 24.5.1. Basic Point Rasterization

Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's $(x_f,y_f)$. 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 pixel center (i.e. at the half-integer) or the location of a sample, and $(x_f,y_f)$ is the exact, unrounded framebuffer coordinate of the vertex for the point. When rasterizationSamples is VK_SAMPLE_COUNT_1_BIT, the pixel center **must** be used.

## 24.6. Line Segments

A line is drawn by generating a set of fragments overlapping a rectangle centered on the line segment. Each line segment has an associated width that controls the width of that rectangle.

The line width is set by the lineWidth property of VkPipelineRasterizationStateCreateInfo in 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                                 commandBuffer,
    float                                           lineWidth);
```

- commandBuffer is the command buffer into which the command will be recorded.
- lineWidth is the width of rasterized line segments.

### Valid Usage

- The currently 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<br>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.

## 24.6.1. Basic Line Segment Rasterization

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 pixel center of a fragment or the location of a sample within the fragment. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the pixel 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 **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.

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 12. Non strict lines*

## 24.7. 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.

### 24.7.1. 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 ith vertex of the n-vertex polygon (vertices are numbered starting at zero for the purposes of this computation) and i ⊕ 1 is (i + 1) mod n.

The interpretation of the sign of a is determined by the VkPipelineRasterizationStateCreateInfo ::`frontFace` property of the currently active pipeline, which takes the following values:

```
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
} VkFrontFace;
```

If `frontFace` is set to `VK_FRONT_FACE_COUNTER_CLOCKWISE`, a triangle with positive area is considered

front-facing. If it is set to VK_FRONT_FACE_CLOCKWISE, 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 setting of the VkPipelineRasterizationStateCreateInfo::cullMode property of the currently active pipeline, which takes the following values:

```
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,
} VkCullModeFlagBits;
```

If the cullMode is set to VK_CULL_MODE_NONE no triangles are discarded, if it is set to VK_CULL_MODE_FRONT_BIT front-facing triangles are discarded, if it is set to VK_CULL_MODE_BACK_BIT then back-facing triangles are discarded and if it is set to VK_CULL_MODE_FRONT_AND_BACK then all triangles are discarded. Following culling, fragments are produced for any triangles which have not been discarded.

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 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:

$$a = \frac{\mathrm{A}(p p_b p_c)}{\mathrm{A}(p_a p_b p_c)}, \quad b = \frac{\mathrm{A}(p p_a p_c)}{\mathrm{A}(p_a p_b p_c)}, \quad c = \frac{\mathrm{A}(p p_a p_b)}{\mathrm{A}(p_a p_b p_c)},$$

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:

$$f = \frac{a f_a / w_a + b f_b / w_b + c f_c / w_c}{a / w_a + b / w_b + c / w_c}$$

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 a pixel center or the location of a sample. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the pixel 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.

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, $0 \le a_i \le 1$ 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).

### 24.7.2. Polygon Mode

The method of rasterization for polygons is determined by the VkPipelineRasterizationStateCreateInfo::`polygonMode` property of the currently active pipeline,

which takes the following values:

```
typedef enum VkPolygonMode {
    VK_POLYGON_MODE_FILL = 0,
    VK_POLYGON_MODE_LINE = 1,
    VK_POLYGON_MODE_POINT = 2,
} VkPolygonMode;
```

The polygonMode selects which method of rasterization is used for polygons. If polygonMode is VK_POLYGON_MODE_POINT, then the vertices of polygons are treated, for rasterization purposes, as if they had been drawn as points. VK_POLYGON_MODE_LINE causes polygon edges to be drawn as line segments. VK_POLYGON_MODE_FILL causes polygons to render using the polygon rasterization rules in this section.

Note that 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.

### 24.7.3. 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                                 commandBuffer,
    float                                           depthBiasConstantFactor,
    float                                           depthBiasClamp,
    float                                           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 no depth buffer is present, r is undefined.

The bias value o for a polygon is

$$o = \begin{cases} m \times depthBiasSlopeFactor + r \times depthBiasConstantFactor & depthBiasClamp = 0 \, or \, NaN \\ \min(m \times depthBiasSlopeFactor + r \times depthBiasConstantFactor, depthBiasClamp) & depthBiasClamp > 0 \\ \max(m \times depthBiasSlopeFactor + r \times depthBiasConstantFactor, depthBiasClamp) & depthBiasClamp < 0 \end{cases}$$

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. Fragment depth values are clamped even when the depth buffer uses a floating-point representation.

## Valid Usage

- The currently 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<br>Secondary | Both | Graphics | |

# Chapter 25. 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.

## 25.1. 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 and sample mask generation are both 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)
- 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
- Sample counting for occlusion queries

## 25.2. 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
- pNext **must** be NULL
- flags **must** be 0
- discardRectangleMode **must** be a valid VkDiscardRectangleModeEXT value

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                             commandBuffer,
    uint32_t                                    firstDiscardRectangle,
    uint32_t                                    discardRectangleCount,
    const VkRect2D*                             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 currently bound graphics pipeline **must** have been created with the VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT dynamic state enabled
- firstDiscardRectangle **must** be less than VkPhysicalDeviceDiscardRectanglePropertiesEXT.maxDiscardRectangles
- The sum of firstDiscardRectangle and discardRectangleCount **must** be between 1 and VkPhysicalDeviceDiscardRectanglePropertiesEXT::maxDiscardRectangles, inclusive
- pDiscardRectangles **must** be a pointer to an array of discardRectangleCount valid VkRect2D structures
- The x and y members of offset in VkRect2D **must** be greater than or equal to 0
- Evaluation of (offset.x + extent.width) in VkRect2D **must** not cause a signed integer addition overflow
- Evaluation of (offset.y + extent.height) in VkRect2D **must** not cause a signed integer addition overflow

## Valid Usage (Implicit)

- commandBuffer **must** be a valid VkCommandBuffer handle
- pDiscardRectangles **must** be a 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<br>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 ≤ $x_f$ < offset.x extent.width and offset.y ≤ $y_f$ < 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 VkDiscardRectangleModeEXT mode enum is defined as:

```
typedef enum VkDiscardRectangleModeEXT {
    VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT = 0,
    VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT = 1,
} VkDiscardRectangleModeEXT;
```

## 25.3. Scissor Test

The scissor test determines if a fragment's framebuffer coordinates ($x_f$,$y_f$) 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                             commandBuffer,
    uint32_t                                    firstScissor,
    uint32_t                                    scissorCount,
    const VkRect2D*                             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, `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 currently 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` **must** be greater than or equal to `0`
- Evaluation of (`offset.x` + `extent.width`) **must** not cause a signed integer addition overflow
- Evaluation of (`offset.y` + `extent.height`) **must** not cause a signed integer addition overflow

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pScissors` **must** be a 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<br>Secondary | Both | Graphics | |

If `offset.x` ≤ $x_f$ < `offset.x` `extent.width` and `offset.y` ≤ $y_f$ < `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 only apply 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.

## 25.4. 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 × 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;
```

## 25.5. Early Fragment Test Mode

The depth bounds test, stencil test, depth 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.

## 25.6. 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 pixel sample, rather than just once for each fragment. Stencil and depth operations are performed for a pixel 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.

## 25.7. 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 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 alphaToCoverageEnable is enabled, a temporary coverage value 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 pixel locations.

*Note*

Using different algorithms at different pixel location **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.

## 25.8. 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

## 25.9. 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 $(x_f,y_f)$ (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                                 commandBuffer,
    float                                           minDepthBounds,
    float                                           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 currently bound graphics pipeline **must** have been created with the `VK_DYNAMIC_STATE_DEPTH_BOUNDS` dynamic state enabled
- `minDepthBounds` **must** be between `0.0` and `1.0`, inclusive
- `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<br>Secondary | Both | Graphics | |

If `minDepthBounds` $\leq z_a \leq$ `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.

## 25.10. 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 with 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 the action performed on samples that fail the stencil test.
- `passOp` is the action performed on samples that pass both the depth and stencil tests.
- `depthFailOp` is the action performed on samples that pass the stencil test and fail the depth test.
- `compareOp` is 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                             commandBuffer,
    VkStencilFaceFlags                          faceMask,
    uint32_t                                    compareMask);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask specifying the set of stencil state for which to update the compare mask. Bits which **can** be set include:

  ```
  typedef enum VkStencilFaceFlagBits {
      VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
      VK_STENCIL_FACE_BACK_BIT = 0x00000002,
      VK_STENCIL_FRONT_AND_BACK = 0x00000003,
  } VkStencilFaceFlagBits;
  ```

  - `VK_STENCIL_FACE_FRONT_BIT` indicates that only the front set of stencil state is updated.
  - `VK_STENCIL_FACE_BACK_BIT` indicates that only the back set of stencil state is updated.
  - `VK_STENCIL_FRONT_AND_BACK` is the combination of `VK_STENCIL_FACE_FRONT_BIT` and `VK_STENCIL_FACE_BACK_BIT` and indicates that both sets of stencil state are updated.
- `compareMask` is the new value to use as the stencil compare mask.

### Valid Usage

- The currently 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<br>Secondary | Both | Graphics | |

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 currently 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<br>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                                 commandBuffer,
    VkStencilFaceFlags                              faceMask,
    uint32_t                                        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 currently 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<br>Secondary | Both | Graphics | |

`reference` is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison clamps the reference value to $[0,2^s-1]$, where s is the number of bits in the stencil framebuffer attachment. 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.

`compareOp` is a symbolic constant that determines the stencil comparison function:

```
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,
} VkCompareOp;
```

- VK_COMPARE_OP_NEVER: the test never passes.
- VK_COMPARE_OP_LESS: the test passes when R < S.
- VK_COMPARE_OP_EQUAL: the test passes when R = S.
- VK_COMPARE_OP_LESS_OR_EQUAL: the test passes when R ≤ S.
- VK_COMPARE_OP_GREATER: the test passes when R > S.
- VK_COMPARE_OP_NOT_EQUAL: the test passes when R ≠ S.
- VK_COMPARE_OP_GREATER_OR_EQUAL: the test passes when R ≥ S.
- VK_COMPARE_OP_ALWAYS: the test always passes.

As described earlier, the failOp, passOp, and depthFailOp members of VkStencilOpState indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. Each enum is of type VkStencilOp, which is defined as:

```
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,
} VkStencilOp;
```

The possible values are:

- 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.

## 25.11. 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 depth clamping (see Primitive Clipping) 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. If `depthWriteEnable` is `VK_TRUE` the value is written, and if it is `VK_FALSE` the value is not written. 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.

## 25.12. 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, sample mask, alpha to coverage, stencil, and depth tests.

# Chapter 26. The Framebuffer

## 26.1. 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 pixel sample, 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 will be performed with a precision and dynamic range no lower than that used to represent destination components.

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 array of per target attachment states.
- `blendConstants` is 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`
- `flags` **must** be `0`
- If `attachmentCount` is not `0`, `pAttachments` **must** be a pointer to an array of `attachmentCount` valid `VkPipelineColorBlendAttachmentState` structures

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 use to calculate the alpha values to write to the color attachment.
- `colorWriteMask` is a bitmask selecting which of the R, G, B, and/or A components are enabled for writing, as described in the Color Write Mask section.

### 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`

### 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

## 26.1.1. 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,
} VkBlendFactor;
```

The semantics of each enum value is described in the table below:

*Table 28. Blend Factors*

| VkBlendFactor | RGB Blend Factors ($S_r$,$S_g$,$S_b$) or ($D_r$,$D_g$,$D_b$) | Alpha Blend Factor ($S_a$ or $D_a$) |
|---|---|---|
| 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 |

| VkBlendFactor | RGB Blend Factors ($S_r$,$S_g$,$S_b$) or ($D_r$,$D_g$,$D_b$) | Alpha Blend Factor ($S_a$ or $D_a$) |
|---|---|---|
| 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                             commandBuffer,
    const float                                 blendConstants[4]);
```

- commandBuffer is the command buffer into which the command will be recorded.
- blendConstants is 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 currently 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<br>Secondary | Both | Graphics | |

## 26.1.2. 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 hardware 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.

## 26.1.3. 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 operation. The blending operations are selected from the following enum, with RGB and alpha components potentially using different blend

operations:

```
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,
} VkBlendOp;
```

The semantics of each enum value is described in the table below:

*Table 29. Blend Operations*

| VkBlendOp | RGB Components | Alpha Component |
|---|---|---|
| VK_BLEND_OP_ADD | $R = R_{s0} \times S_r + R_d \times D_r$<br>$G = G_{s0} \times S_g + G_d \times D_g$<br>$B = B_{s0} \times S_b + B_d \times D_b$ | $A = A_{s0} \times S_a + A_d \times D_a$ |
| VK_BLEND_OP_SUBTRACT | $R = R_{s0} \times S_r - R_d \times D_r$<br>$G = G_{s0} \times S_g - G_d \times D_g$<br>$B = B_{s0} \times S_b - B_d \times D_b$ | $A = A_{s0} \times S_a - A_d \times D_a$ |
| VK_BLEND_OP_REVERSE_SUBTRACT | $R = R_d \times D_r - R_{s0} \times S_r$<br>$G = G_d \times D_g - G_{s0} \times S_g$<br>$B = B_d \times D_b - B_{s0} \times S_b$ | $A = A_d \times D_a - A_{s0} \times S_a$ |
| VK_BLEND_OP_MIN | $R = \min(R_{s0},R_d)$<br>$G = \min(G_{s0},G_d)$<br>$B = \min(B_{s0},B_d)$ | $A = \min(A_{s0},A_d)$ |
| VK_BLEND_OP_MAX | $R = \max(R_{s0},R_d)$<br>$G = \max(G_{s0},G_d)$<br>$B = \max(B_{s0},B_d)$ | $A = \max(A_{s0},A_d)$ |

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 “KHR_DF_TRANSFER_SRGB” 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 “KHR_DF_TRANSFER_SRGB” 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.

## 26.2. 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,
} VkLogicOp;
```

The logical operations supported by Vulkan are summarized in the following table in which

- ¬ is bitwise invert,
- ∧ is bitwise and,
- ∨ is bitwise or,
- ⊕ 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 30. Logical Operations*

| Mode | Operation |
|---|---|
| `VK_LOGIC_OP_CLEAR` | 0 |
| `VK_LOGIC_OP_AND` | s ∧ d |
| `VK_LOGIC_OP_AND_REVERSE` | s ∧ ¬ d |
| `VK_LOGIC_OP_COPY` | s |
| `VK_LOGIC_OP_AND_INVERTED` | ¬ s ∧ d |
| `VK_LOGIC_OP_NO_OP` | d |
| `VK_LOGIC_OP_XOR` | s ⊕ d |
| `VK_LOGIC_OP_OR` | s ∨ d |
| `VK_LOGIC_OP_NOR` | ¬ (s ∨ d) |
| `VK_LOGIC_OP_EQUIVALENT` | ¬ (s ⊕ d) |
| `VK_LOGIC_OP_INVERT` | ¬ d |
| `VK_LOGIC_OP_OR_REVERSE` | s ∨ ¬ d |
| `VK_LOGIC_OP_COPY_INVERTED` | ¬ s |
| `VK_LOGIC_OP_OR_INVERTED` | ¬ s ∨ d |
| `VK_LOGIC_OP_NAND` | ¬ (s ∧ d) |
| `VK_LOGIC_OP_SET` | 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.

## 26.3. Color Write Mask

The `colorWriteMask` member of VkPipelineColorBlendAttachmentState determines whether the final color values R, G, B and A are written to the framebuffer attachment. `colorWriteMask` is any combination of the following bits:

```
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,
} VkColorComponentFlagBits;
```

If `VK_COLOR_COMPONENT_R_BIT` is set, then the R value is written to color attachment for the appropriate sample, otherwise the value in memory is unmodified. The `VK_COLOR_COMPONENT_G_BIT`, `VK_COLOR_COMPONENT_B_BIT`, and `VK_COLOR_COMPONENT_A_BIT` bits similarly control writing of the G, B, and A values.

The `colorWriteMask` is applied regardless of whether blending is enabled.

# Chapter 27. 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 currently 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                                 commandBuffer,
    uint32_t                                        groupCountX,
    uint32_t                                        groupCountY,
    uint32_t                                        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

- `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]
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_COMPUTE`, with a `VkPipelineLayout` that is compatible for set *n*, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- A valid compute pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_COMPUTE`
- For each push constant that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_COMPUTE`, with a `VkPipelineLayout` that is compatible for push constants with the one used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility
- If any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` uses unnormalized coordinates, it **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 any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` uses unnormalized coordinates, it **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 any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` uses unnormalized coordinates, it **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 any shader stage in the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound

descriptor set

- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** be of a format which supports cubic filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** not have a VkImageViewType of VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY

## 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<br>Secondary | Outside | Compute | Compute |

To record an indirect command dispatch, call:

```
void vkCmdDispatchIndirect(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                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 `buffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- For each set *n* that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE`, a descriptor set **must** have been bound to *n* at `VK_PIPELINE_BIND_POINT_COMPUTE`, with a `VkPipelineLayout` that is compatible for set *n*, 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 currently bound `VkPipeline` object, specified via `vkCmdBindPipeline`
- A valid compute pipeline **must** be bound to the current command buffer with `VK_PIPELINE_BIND_POINT_COMPUTE`
- `buffer` **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set
- `offset` **must** be a multiple of `4`
- The sum of `offset` and the size of `VkDispatchIndirectCommand` **must** be less than or equal to the size of `buffer`
- For each push constant that is statically used by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE`, a push constant value **must** have been set for `VK_PIPELINE_BIND_POINT_COMPUTE`, with a `VkPipelineLayout` that is compatible for push constants with the one used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility
- If any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` uses unnormalized coordinates, it **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 any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` uses unnormalized coordinates, it **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 any `VkSampler` object that is accessed from a shader by the `VkPipeline` currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` uses unnormalized coordinates, it **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 any shader stage in the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` accesses a uniform buffer, it **must** not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the `VkPipeline` object currently bound to `VK_PIPELINE_BIND_POINT_COMPUTE` accesses a storage buffer, it **must** not access values outside of the range of that buffer specified in the currently bound

descriptor set

- Any VkImageView being sampled with VK_FILTER_LINEAR as a result of this command **must** be of a format which supports linear filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** be of a format which supports cubic filtering, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG flag in VkFormatProperties::linearTilingFeatures (for a linear image) or VkFormatProperties::optimalTilingFeatures(for an optimally tiled image) returned by vkGetPhysicalDeviceFormatProperties
- Any VkImageView being sampled with VK_FILTER_CUBIC_IMG as a result of this command **must** not have a VkImageViewType of VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY

## 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<br>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 vkCmdDispatchBaseKHX(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    baseGroupX,
    uint32_t                                    baseGroupY,
    uint32_t                                    baseGroupZ,
    uint32_t                                    groupCountX,
    uint32_t                                    groupCountY,
    uint32_t                                    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 vkCmdDispatchBaseKHX(0,0,0,groupCountX,groupCountY,groupCountZ).

## Valid Usage

- All valid usage rules from vkCmdDispatch apply
- `baseGroupX` **must** be less than `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`
- `baseGroupX` **must** be less than `VkPhysicaYDeviceLimits::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 currently bound compute pipeline **must** have been created with the `VK_PIPELINE_CREATE_DISPATCH_BASE_KHX` 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<br>Secondary | Outside | Compute | |

# Chapter 28. 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 an `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.

## 28.1. 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` points to an instance of the VkDeviceGeneratedCommandsFeaturesNVX structure, that will be filled with returned information.
- `pLimits` points to an instance of the VkDeviceGeneratedCommandsLimitsNVX structure, that will be filled with returned information.

### Valid Usage (Implicit)

- physicalDevice **must** be a valid VkPhysicalDevice handle
- pFeatures **must** be a pointer to a VkDeviceGeneratedCommandsFeaturesNVX structure
- pLimits **must** be a 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 indicates 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

## 28.2. 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.

### 28.2.1. 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 an instance of the VkObjectTableCreateInfoNVX structure containing parameters affecting creation of the table.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pObjectTable points 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 pointer to a valid `VkObjectTableCreateInfoNVX` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pObjectTable` **must** be a 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 an array of VkObjectEntryTypeNVX providing the entry type of a given configuration.
- `pObjectEntryCounts` is an array of counts how many objects can be registered in the table.
- `pObjectEntryUsageFlags` is an array of bitmasks describing the binding usage of the entry. See VkObjectEntryUsageFlagBitsNVX below for a description of the supported bits.
- `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.

Types which **can** be set in pObjectEntryTypes are:

```
typedef enum VkObjectEntryTypeNVX {
    VK_OBJECT_ENTRY_DESCRIPTOR_SET_NVX = 0,
    VK_OBJECT_ENTRY_PIPELINE_NVX = 1,
    VK_OBJECT_ENTRY_INDEX_BUFFER_NVX = 2,
    VK_OBJECT_ENTRY_VERTEX_BUFFER_NVX = 3,
    VK_OBJECT_ENTRY_PUSH_CONSTANT_NVX = 4,
} VkObjectEntryTypeNVX;
```

- VK_OBJECT_ENTRY_DESCRIPTOR_SET_NVX indicates a VkDescriptorSet resource entry that is registered via VkObjectTableDescriptorSetEntryNVX.
- VK_OBJECT_ENTRY_PIPELINE_NVX indicates a VkPipeline resource entry that is registered via VkObjectTablePipelineEntryNVX.
- VK_OBJECT_ENTRY_INDEX_BUFFER_NVX indicates a VkBuffer resource entry that is registered via VkObjectTableIndexBufferEntryNVX.
- VK_OBJECT_ENTRY_VERTEX_BUFFER_NVX indicates a VkBuffer resource entry that is registered via VkObjectTableVertexBufferEntryNVX.
- VK_OBJECT_ENTRY_PUSH_CONSTANT_NVX indicates the resource entry is registered via VkObjectTablePushConstantEntryNVX.

Bits which **can** be set in pObjectEntryUsageFlags are:

```
typedef enum VkObjectEntryUsageFlagBitsNVX {
    VK_OBJECT_ENTRY_USAGE_GRAPHICS_BIT_NVX = 0x00000001,
    VK_OBJECT_ENTRY_USAGE_COMPUTE_BIT_NVX = 0x00000002,
} VkObjectEntryUsageFlagBitsNVX;
```

- VK_OBJECT_ENTRY_USAGE_GRAPHICS_BIT_NVX indicates that the resource is bound to VK_PIPELINE_BIND_POINT_GRAPHICS

- VK_OBJECT_ENTRY_USAGE_COMPUTE_BIT_NVX indicates that the resource is bound to VK_PIPELINE_BIND_POINT_COMPUTE

### 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 pointer to an array of objectCount valid VkObjectEntryTypeNVX values
- pObjectEntryCounts **must** be a pointer to an array of objectCount uint32_t values
- pObjectEntryUsageFlags **must** be a 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

To destroy an object table, call:

```
void vkDestroyObjectTableNVX(
    VkDevice                                    device,
    VkObjectTableNVX                            objectTable,
    const VkAllocationCallbacks*                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 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

## 28.2.2. 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                                    device,
    VkObjectTableNVX                            objectTable,
    uint32_t                                    objectCount,
    const VkObjectTableEntryNVX* const*         ppObjectTableEntries,
    const uint32_t*                             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 struct 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 pointer to an array of `objectCount` valid `VkObjectTableEntryNVX` structures
- `pObjectIndices` **must** be a 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 {
    VkObjectEntryTypeNVX          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 VkObjectEntryTypeNVX value
- `flags` **must** be a valid combination of VkObjectEntryUsageFlagBitsNVX values
- `flags` **must** not be `0`

```
typedef struct VkObjectTablePipelineEntryNVX {
    VkObjectEntryTypeNVX          type;
    VkObjectEntryUsageFlagsNVX    flags;
    VkPipeline                    pipeline;
} VkObjectTablePipelineEntryNVX;
```

- `pipeline` specifies the `VkPipeline` that this resource entry references.

### Valid Usage

- `type` **must** be `VK_OBJECT_ENTRY_PIPELINE_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`
- `pipeline` **must** be a valid `VkPipeline` handle

```
typedef struct VkObjectTableDescriptorSetEntryNVX {
    VkObjectEntryTypeNVX          type;
    VkObjectEntryUsageFlagsNVX    flags;
    VkPipelineLayout              pipelineLayout;
    VkDescriptorSet               descriptorSet;
} VkObjectTableDescriptorSetEntryNVX;
```

- `layout` 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_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_VERTEX_BUFFER_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
- buffer **must** be a valid VkBuffer handle

```
typedef struct VkObjectTableIndexBufferEntryNVX {
    VkObjectEntryTypeNVX          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_INDEX_BUFFER_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`
- `buffer` **must** be a valid `VkBuffer` handle
- `indexType` **must** be a valid VkIndexType value

```
typedef struct VkObjectTablePushConstantEntryNVX {
    VkObjectEntryTypeNVX          type;
    VkObjectEntryUsageFlagsNVX    flags;
    VkPipelineLayout              pipelineLayout;
    VkShaderStageFlags            stageFlags;
} VkObjectTablePushConstantEntryNVX;
```

- `layout` specifies the `VkPipelineLayout` that pushconstants using this `objectIndex` are used with.
- `descriptorSet` stageFlags the `VkShaderStageFlags` that pushconstants using this `objectIndex` are used with.

### Valid Usage

- `type` **must** be `VK_OBJECT_ENTRY_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`

Using 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 pointer to an array of `objectCount` valid VkObjectEntryTypeNVX values
- `pObjectIndices` **must** be a 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`

## 28.3. 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)
```

### 28.3.1. 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.

The following tokens exist:

```
typedef enum VkIndirectCommandsTokenTypeNVX {
    VK_INDIRECT_COMMANDS_TOKEN_PIPELINE_NVX = 0,
    VK_INDIRECT_COMMANDS_TOKEN_DESCRIPTOR_SET_NVX = 1,
    VK_INDIRECT_COMMANDS_TOKEN_INDEX_BUFFER_NVX = 2,
    VK_INDIRECT_COMMANDS_TOKEN_VERTEX_BUFFER_NVX = 3,
    VK_INDIRECT_COMMANDS_TOKEN_PUSH_CONSTANT_NVX = 4,
    VK_INDIRECT_COMMANDS_TOKEN_DRAW_INDEXED_NVX = 5,
    VK_INDIRECT_COMMANDS_TOKEN_DRAW_NVX = 6,
    VK_INDIRECT_COMMANDS_TOKEN_DISPATCH_NVX = 7,
} VkIndirectCommandsTokenTypeNVX;
```

*Table 31. Supported indirect command tokens*

| Token type | Equivalent command |
|---|---|
| VK_INDIRECT_COMMANDS_TOKEN_PIPELINE_NVX | vkCmdBindPipeline |
| VK_INDIRECT_COMMANDS_TOKEN_DESCRIPTOR_SET_NVX | vkCmdBindDescriptorSets |
| VK_INDIRECT_COMMANDS_TOKEN_INDEX_BUFFER_NVX | vkCmdBindIndexBuffer |

| Token type | Equivalent command |
|---|---|
| VK_INDIRECT_COMMANDS_TOKEN_VERTEX_BUFFER_NVX | vkCmdBindVertexBuffers |
| VK_INDIRECT_COMMANDS_TOKEN_PUSH_CONSTANT_NVX | vkCmdPushConstants |
| VK_INDIRECT_COMMANDS_TOKEN_DRAW_INDEXED_NVX | vkCmdDrawIndexedIndirect |
| VK_INDIRECT_COMMANDS_TOKEN_DRAW_NVX | vkCmdDrawIndirect |
| VK_INDIRECT_COMMANDS_TOKEN_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 squence. These argumetns can be written by the device.
- `offset` specified 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 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_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_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_PUSH_CONSTANT_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];

        vkCmdPushConstants(cmd,
          objectTable.pushconstants[ object ].layout,
          objectTable.pushconstants[ object ].stageFlags,
          indirectCommandsLayout.pTokens[ c ].bindingUnit, indirectCommandsLayout.
pTokens[c].dynamicCount, data + 1);
      break;

      VK_INDIRECT_COMMANDS_TOKEN_INDEX_BUFFER_NVX:
        size_t   s tride  = 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_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_DRAW_INDEXED_NVX:
        vkCmdDrawIndexedIndirect(cmd,
          input.buffer,
          sizeof(VkDrawIndexedIndirectCommand) * i + input.offset, 1, 0);
      break;

      VK_INDIRECT_COMMANDS_TOKEN_DRAW_NVX:
        vkCmdDrawIndirect(cmd,
          input.buffer,
          sizeof(VkDrawIndirectCommand) * i  + input.offset, 1, 0);
      break;
```

```
      VK_INDIRECT_COMMANDS_TOKEN_DISPATCH_NVX:
        vkCmdDispatchIndirect(cmd,
          input.buffer,
          sizeof(VkDispatchIndirectCommand) * i  + input.offset);
      break;
    }
  }
}
```

### 28.3.2. 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 object table.
- `pCreateInfo` is a pointer to an instance of the `VkObjectTableCreateInfoNVX` structure containing parameters affecting creation of the table.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pIndirectCommandsLayout` points 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 pointer to a valid `VkIndirectCommandsLayoutCreateInfoNVX` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pIndirectCommandsLayout` **must** be a 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 providing usage hints of this layout. See VkIndirectCommandsLayoutUsageFlagBitsNVX below for a description of the supported bits.
- tokenCount is the length of the individual command sequnce.
- pTokens is an array describing each command token in detail. See VkIndirectCommandsTokenTypeNVX and VkIndirectCommandsLayoutTokenNVX below for details.

Bits which **can** be set in flags 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,
} VkIndirectCommandsLayoutUsageFlagBitsNVX;
```

- VK_INDIRECT_COMMANDS_LAYOUT_USAGE_UNORDERED_SEQUENCES_BIT_NVX indicates 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 indicates that there is likely a high difference between allocated number of sequences and actually used.
- VK_INDIRECT_COMMANDS_LAYOUT_USAGE_EMPTY_EXECUTIONS_BIT_NVX indicates that there is 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 indicates that the input data for the sequences is not implicitly indexed from 0..sequencesUsed but a user provided VkBuffer encoding the index is provided.

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_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_DRAW_NVX`, `VK_INDIRECT_COMMANDS_TOKEN_DRAW_INDEXED_NVX`, `VK_INDIRECT_COMMANDS_TOKEN_DISPATCH_NVX`).
- The content of `pTokens` **must** include at least one work provoking token.

### 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 pointer to an array of `tokenCount` valid `VkIndirectCommandsLayoutTokenNVX` structures
- `tokenCount` **must** be greater than `0`

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 pointer to a valid `VkAllocationCallbacks` structure
- `indirectCommandsLayout` **must** have been created, allocated, or retrieved from `device`

## 28.4. 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 an instance of the vkCmdReserveSpaceForCommandsNVX 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 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<br>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 an instance of the VkCmdProcessCommandsInfoNVX structure containing parameters affecting the processing of commands.

## Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pProcessCommandsInfo` **must** be a 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<br>Secondary | Inside | Graphics<br>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` 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 `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 `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 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 **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 29. 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.

## 29.1. 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.
- 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 is not **required** for sparse 3D and MSAA images and is 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.

## 29.2. 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 pixel 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.

### 29.2.1. 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.

## 29.3. 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.

## 29.4. 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 level-of-detail 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 pixel 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.

### 29.4.1. 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 hardware 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 of causing a hardware fault.
>
> Given that reads of unbound regions are undefined in this scenario, implementations **may** use the same physical memory for unbound regions of multiple resources within the same process.

### 29.4.2. 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 13. 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 pixel 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 14. 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 15. 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 pixels 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 16. 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 pixels 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.

### 29.4.3. 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 pixels 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 pixel size is the size of the compressed texel block (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 32. Standard Sparse Image Block Shapes (Single Sample)*

| PIXEL SIZE (bits) | Block Shape (2D) | Block Shape (3D) |
|---|---|---|
| **8-Bit** | 256 × 256 × 1 | 64 × 32 × 32 |
| **16-Bit** | 256 × 128 × 1 | 32 × 32 × 32 |
| **32-Bit** | 128 × 128 × 1 | 32 × 32 × 16 |
| **64-Bit** | 128 × 64 × 1 | 32 × 16 × 16 |
| **128-Bit** | 64 × 64 × 1 | 16 × 16 × 16 |

*Table 33. Standard Sparse Image Block Shapes (MSAA)*

| PIXEL SIZE (bits) | Block Shape (2X) | Block Shape (4X) | Block Shape (8X) | Block Shape (16X) |
|---|---|---|---|---|
| **8-Bit** | 128 × 256 × 1 | 128 × 128 × 1 | 64 × 128 × 1 | 64 × 64 × 1 |
| **16-Bit** | 128 × 128 × 1 | 128 × 64 × 1 | 64 × 64 × 1 | 64 × 32 × 1 |
| **32-Bit** | 64 × 128 × 1 | 64 × 64 × 1 | 32 × 64 × 1 | 32 × 32 × 1 |
| **64-Bit** | 64 × 64 × 1 | 64 × 32 × 1 | 32 × 32 × 1 | 32 × 16 × 1 |
| **128-Bit** | 32 × 64 × 1 | 32 × 32 × 1 | 16 × 32 × 1 | 16 × 16 × 1 |

Implementations that support the standard sparse image block shape for all applicable formats **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.

### 29.4.4. 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.

### 29.4.5. 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 17. 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 pixels to sparse blocks.

In the figure above the depth, stencil, and metadata aspects all have unique sparse properties. The per-pixel stencil data is ¼ the size of the depth data, hence the stencil sparse blocks include 4 × the number of pixels. 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.

## 29.5. 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.

## 29.6. Sparse Resource Implementation Guidelines

This section is Informative. It is included to aid in implementors' understanding of sparse resources.

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. Hardware that supports 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 the cache implementation of the hardware this **may** not always be possible.

Hardware that does not fault, but does not guarantee correct read values will not require dummy pages, but also **must** not support `residencyNonResidentStrict`.

Hardware 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).

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 hardware requires “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 does not require any sophisticated logic to determine what device virtual address to bind. The cost is that the application **can** 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 pixel locations to determine which device virtual addresses to bind, it is impossible to bind device virtual address holes with this operation.

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 denotes 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.

## 29.7. 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

### 29.7.1. 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.

## 29.7.2. 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.

### 29.7.3. 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 of 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 specifying additional information about the sparse resource. Bits which **can** be set include:

```
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,
} 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 first mip level whose dimensions are not integer multiples of the corresponding dimensions of the sparse image block begins the mip tail region.
  - If VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT is set, 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 pixel format.

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 pixel as defined in VkSampleCountFlagBits.
- usage is a bitmask describing the intended usage of the image.
- tiling is the tiling arrangement of the data elements 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 pointer to a uint32_t value
- If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties **must** be a pointer to an array of pPropertyCount VkSparseImageFormatProperties structures

vkGetPhysicalDeviceSparseImageFormatProperties2KHR returns an array of VkSparseImageFormatProperties2KHR. 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 vkGetPhysicalDeviceSparseImageFormatProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    const VkPhysicalDeviceSparseImageFormatInfo2KHR* pFormatInfo,
    uint32_t*                                   pPropertyCount,
    VkSparseImageFormatProperties2KHR*          pProperties);
```

- `physicalDevice` is the physical device from which to query the sparse image capabilities.
- `pFormatInfo` is a pointer to a structure of type VkPhysicalDeviceSparseImageFormatInfo2KHR 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 VkSparseImageFormatProperties2KHR structures.

`vkGetPhysicalDeviceSparseImageFormatProperties2KHR` behaves identically to vkGetPhysicalDeviceSparseImageFormatProperties, with the ability to return extended information by adding extension structures to its `pNext` chain.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pFormatInfo` **must** be a pointer to a valid `VkPhysicalDeviceSparseImageFormatInfo2KHR` structure
- `pPropertyCount` **must** be a pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a pointer to an array of `pPropertyCount` `VkSparseImageFormatProperties2KHR` structures

The `VkPhysicalDeviceSparseImageFormatInfo2KHR` structure is defined as:

```
typedef struct VkPhysicalDeviceSparseImageFormatInfo2KHR {
    VkStructureType          sType;
    const void*              pNext;
    VkFormat                 format;
    VkImageType              type;
    VkSampleCountFlagBits    samples;
    VkImageUsageFlags        usage;
    VkImageTiling            tiling;
} 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 pixel as defined in VkSampleCountFlagBits.
- usage is a bitmask describing the intended usage of the image.
- tiling is the tiling arrangement of the data elements 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_KHR
- 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 VkSparseImageFormatProperties2KHR structure is defined as:

```
typedef struct VkSparseImageFormatProperties2KHR {
    VkStructureType                 sType;
    void*                           pNext;
    VkSparseImageFormatProperties   properties;
} VkSparseImageFormatProperties2KHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to an extension-specific structure.
- properties is a structure of type VkSparseImageFormatProperties which is populated with the same values as in vkGetPhysicalDeviceSparseImageFormatProperties.

## 29.7.4. 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).

### 29.7.5. 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 denotes 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 denotes 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 pixel units. For aspect VK_IMAGE_ASPECT_METADATA_BIT, all dimensions will be zero pixels. 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 pixel 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 hardware 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 pointer to a uint32_t value
- If the value referenced by pSparseMemoryRequirementCount is not 0, and pSparseMemoryRequirements is not NULL, pSparseMemoryRequirements **must** be a pointer to an array of pSparseMemoryRequirementCount VkSparseImageMemoryRequirements structures
- image **must** have been created, allocated, or retrieved from device

### 29.7.6. 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. Access to resources that are bound to memory objects that have been freed will result in undefined behavior, potentially including application termination.
>
> Implementations **must** ensure that no access to physical memory owned by the system or another process will occur in this scenario. In other words, accessing resources bound to freed memory **may** result in application termination, but **must** not result in system termination or in reading non-process-accessible memory.

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. Accessing memory in a range while it is being rebound results in undefined behavior. It is valid to 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 specifying usage of the binding operation. Bits which **can** be set include:

```
typedef enum VkSparseMemoryBindFlagBits {
    VK_SPARSE_MEMORY_BIND_METADATA_BIT = 0x00000001,
} VkSparseMemoryBindFlagBits;
```

  - `VK_SPARSE_MEMORY_BIND_METADATA_BIT` indicates that the memory being bound is only for the metadata aspect.

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`

### 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

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 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 array of `VkSparseMemoryBind` structures.

### Valid Usage

- For any given element of `pBinds`, if the `flags` member of that element 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 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 array of VkSparseImageMemoryBind structures

### Valid Usage (Implicit)

- image **must** be a valid VkImage handle
- pBinds **must** be a 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 aspectMask 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

### 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                                     fence);
```

- `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 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
- Any given element of the pSignalSemaphores member of any element of pBindInfo **must** be unsignaled when the semaphore signal operation it defines is executed on the device
- When a semaphore unsignal operation defined by any element of the pWaitSemaphores member of any element of pBindInfo executes on queue, no other queue **must** be waiting on the same semaphore.
- All elements of the pWaitSemaphores member of all elements of pBindInfo **must** be semaphores that are signaled, or have semaphore signal operations previously submitted for execution.

## Valid Usage (Implicit)

- queue **must** be a valid VkQueue handle
- If bindInfoCount is not 0, pBindInfo **must** be a 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 **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`[].pWaitSemaphores[] **must** be externally synchronized
- Host access to `pBindInfo`[].pSignalSemaphores[] **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                             sType;
    const void*                                 pNext;
    uint32_t                                    waitSemaphoreCount;
    const VkSemaphore*                          pWaitSemaphores;
    uint32_t                                    bufferBindCount;
    const VkSparseBufferMemoryBindInfo*         pBufferBinds;
    uint32_t                                    imageOpaqueBindCount;
    const VkSparseImageOpaqueMemoryBindInfo*    pImageOpaqueBinds;
    uint32_t                                    imageBindCount;
    const VkSparseImageMemoryBindInfo*          pImageBinds;
    uint32_t                                    signalSemaphoreCount;
    const VkSemaphore*                          pSignalSemaphores;
} 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 (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_BIND_SPARSE_INFO
- pNext **must** be NULL or a pointer to a valid instance of VkDeviceGroupBindSparseInfoKHX
- If waitSemaphoreCount is not 0, pWaitSemaphores **must** be a pointer to an array of waitSemaphoreCount valid VkSemaphore handles
- If bufferBindCount is not 0, pBufferBinds **must** be a pointer to an array of bufferBindCount valid VkSparseBufferMemoryBindInfo structures
- If imageOpaqueBindCount is not 0, pImageOpaqueBinds **must** be a pointer to an array of imageOpaqueBindCount valid VkSparseImageOpaqueMemoryBindInfo structures
- If imageBindCount is not 0, pImageBinds **must** be a pointer to an array of imageBindCount valid VkSparseImageMemoryBindInfo structures
- If signalSemaphoreCount is not 0, pSignalSemaphores **must** be a pointer to an array of signalSemaphoreCount valid VkSemaphore handles
- Both of the elements of pSignalSemaphores, and the elements of pWaitSemaphores that are valid handles **must** have been created, allocated, or retrieved from the same VkDevice

If the pNext list of VkBindSparseInfo includes a VkDeviceGroupBindSparseInfoKHX structure, then that structure includes device indices specifying which instance of the resources and memory are bound.

The VkDeviceGroupBindSparseInfoKHX structure is defined as:

```
typedef struct VkDeviceGroupBindSparseInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           resourceDeviceIndex;
    uint32_t           memoryDeviceIndex;
} VkDeviceGroupBindSparseInfoKHX;
```

- 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 that points 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_KHX`

## 29.8. Examples

The following examples illustrate basic creation of sparse images and binding them to physical memory.

### 29.8.1. 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);
```

### 29.8.2. 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 = {};
uint32_t                            sparseRequirementsCount = 0;
VkSparseImageMemoryRequirements*    pSparseReqs = NULL;
VkSparseMemoryBind                  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 |
    VK_IMAGE_USAGE_SAMPLED_BIT,                 // usage
    ...
};
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 30. 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.

## 30.1. 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, which contain 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** #define the appropriate preprocessor symbol prior to including the "vulkan.h" header file. Each platform-specific extension is an instance extension. The application **must** enable instance extensions with `vkCreateInstance` before using them.

## 30.2. 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 struct that contains 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.

### 30.2.1. 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 an instance of the `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` points 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 pointer to a valid `VkAndroidSurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a 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;
    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

## 30.2.2. Mir Platform

To create a VkSurfaceKHR object for a Mir window, call:

```
VkResult vkCreateMirSurfaceKHR(
    VkInstance                                  instance,
    const VkMirSurfaceCreateInfoKHR*            pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to an instance of the VkMirSurfaceCreateInfoKHR 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` points 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 pointer to a valid `VkMirSurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a pointer to a `VkSurfaceKHR` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkMirSurfaceCreateInfoKHR` structure is defined as:

```
typedef struct VkMirSurfaceCreateInfoKHR {
    VkStructureType                 sType;
    const void*                     pNext;
    VkMirSurfaceCreateFlagsKHR      flags;
    MirConnection*                  connection;
    MirSurface*                     mirSurface;
} VkMirSurfaceCreateInfoKHR;
```

- `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` and `surface` are pointers to the `MirConnection` and `MirSurface` for the window to associate the surface with.

### Valid Usage

- `connection` **must** point to a valid `MirConnection`.
- `surface` **must** point to a valid `MirSurface`.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_MIR_SURFACE_CREATE_INFO_KHR`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`

On Mir, 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 supported surface size.

### 30.2.3. 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 an instance of the 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` points 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 pointer to a valid `VkWaylandSurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a 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 undefined (0xFFFFFFFF, 0xFFFFFFFF). 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.

## 30.2.4. 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 an instance of the 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 points 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 pointer to a valid VkWin32SurfaceCreateInfoKHR structure
- If pAllocator is not NULL, pAllocator **must** be a pointer to a valid VkAllocationCallbacks structure
- pSurface **must** be a 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 and hwnd are the Win32 HINSTANCE and 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` are the window size. Therefore, a swapchain's `imageExtent` **must** match the window's size.

## 30.2.5. 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 an instance of the `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` points 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 pointer to a valid `VkXcbSurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a 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 are the window size. Therefore, a swapchain's imageExtent **must** match the window's size.

Some Vulkan functions **may** send protocol over the specified xcb connection when using a swapchain or presentable images created from a VkSurface 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.

### 30.2.6. 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 an instance of the `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` points 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 pointer to a valid `VkXlibSurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a 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` are the window size. Therefore, a swapchain's `imageExtent` **must** match the window's size.

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.

### 30.2.7. 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 an instance of the 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 points 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 pointer to a valid VkIOSSurfaceCreateInfoMVK structure
- If pAllocator is not NULL, pAllocator **must** be a pointer to a valid VkAllocationCallbacks structure
- pSurface **must** be a 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`
- `pView` **must** be a pointer

## 30.2.8. 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 an instance of the `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` points 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 pointer to a valid `VkMacOSSurfaceCreateInfoMVK` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a 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`
- `pView` **must** be a pointer

## 30.2.9. 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 an instance of the `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` points 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` any attempts to create another surface for the same `nn::vi::Layer` and any attempts to connect to the same `nn::vi::Layer` through other platform mechanisms will have undefined results.

The `currentExtent` of a VI surface is always undefined. Applications are expected to choose an appropriate size for the swapchain's `imageExtent` (e.g., by matching the the result of a call to `nn::vi::GetDisplayResolution`).

### Valid Usage (Implicit)

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

## 30.2.10. 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 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

# 30.3. 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.

## 30.3.1. 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 pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a 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 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 tells which transforms are supported by this display. This will contain one or more of the bits from VkSurfaceTransformFlagsKHR.
- 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.

### 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 looses access to `display`. During such periods, operations which require access to the display **must** fail with an approriate 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 pointer to a `Display` value
- `display` **must** be a valid `VkDisplayKHR` handle

### Return Codes

**Success**

- VK_SUCCESS

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 pointer to a Display value
- pDisplay **must** be a 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 pointer to a uint32_t value
- If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties **must** be a 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 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 pointer to a `uint32_t` value
- If the value referenced by `pDisplayCount` is not `0`, and `pDisplays` is not `NULL`, `pDisplays` **must** be a 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 pointer to a uint32_t value
- If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties **must** be a 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.

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.

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 pointer to a valid `VkDisplayModeCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pMode` **must** be a 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

- The width and height members of the visibleRegion member of parameters **must** be greater than 0
- The refreshRate member of parameters **must** be greater than 0

## Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR
- pNext **must** be NULL
- flags **must** be 0

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 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 hardware 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 hardware's individual fields. Not all combinations of values within the offset and extent ranges returned in VkDisplayPlaneCapabilitiesKHR are guaranteed to be supported. Vendors **may** still fail presentation requests that specify unsupported combinations.

### 30.3.2. 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 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 the new power state of the display. Possible values 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,
} VkDisplayPowerStateEXT;
```

- VK_DISPLAY_POWER_STATE_OFF_EXT means the display is powered down.
- VK_DISPLAY_POWER_STATE_SUSPEND_EXT means the display is in a low power mode, but **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 is powered on.

**Valid Usage (Implicit)**

- sType **must** be VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT
- pNext **must** be NULL
- powerState **must** be a valid VkDisplayPowerStateEXT value

### 30.3.3. 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 an instance of the 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 points 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 pointer to a valid `VkDisplaySurfaceCreateInfoKHR` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pSurface` **must** be a 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 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 the transform to apply to the 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 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

Types of alpha blending supported by or used on a display are defined by the bitmask `VkDisplayPlaneAlphaFlagBitsKHR`, which contains the following values:

```
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,
} VkDisplayPlaneAlphaFlagBitsKHR;
```

These values are described as follows:

- `VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR`: The source image will be treated as opaque.
- `VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR`: 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`: 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`: This 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.

## 30.4. 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 pointer to a VkBool32 value

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

## 30.4.1. 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.

## 30.4.2. Mir Platform

To determine whether a queue family of a physical device supports presentation to the Mir compositor, call:

```
VkBool32 vkGetPhysicalDeviceMirPresentationSupportKHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t                                    queueFamilyIndex,
    MirConnection*                              connection);
```

- physicalDevice is the physical device.
- queueFamilyIndex is the queue family index.
- connection is a pointer to the MirConnection, and identifies the desired Mir 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
- `connection` **must** be a pointer to a `MirConnection` value

## 30.4.3. 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 pointer to a `wl_display` value

## 30.4.4. 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

## 30.4.5. 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 pointer to a `xcb_connection_t` value

## 30.4.6. 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 pointer to a `Display` value

## 30.4.7. 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.

## 30.4.8. 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.

### 30.4.9. 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.

## 30.5. Surface Queries

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 an instance of the 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 pointer to a `VkSurfaceCapabilitiesKHR` 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 `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, describing the presentation transforms supported for the surface on the specified device, and at least one bit will be set.
- `currentTransform` is the surface's current transform relative to the presentation engine's natural orientation, as described by VkSurfaceTransformFlagBitsKHR.
- `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, 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 points to an instance of the VkPhysicalDeviceSurfaceInfo2KHR structure, describing the surface and other fixed parameters that would be consumed by vkCreateSwapchainKHR.
- pSurfaceCapabilities points to an instance of the 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 (Implicit)

- physicalDevice **must** be a valid VkPhysicalDevice handle
- pSurfaceInfo **must** be a pointer to a valid VkPhysicalDeviceSurfaceInfo2KHR structure
- pSurfaceCapabilities **must** be a 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 strucutre 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 vkGetPhysicalDeviceSurfaceCapabilities, with sType and pNext added for extensibility.

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR
- pNext **must** be NULL
- surface **must** be a valid VkSurfaceKHR handle

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 structure of type VkSurfaceCapabilitiesKHR describing the capabilities of

the specified surface.

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.

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 an instance of the 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 its pNext chain.

### Valid Usage (Implicit)

- physicalDevice **must** be a valid VkPhysicalDevice handle
- surface **must** be a valid VkSurfaceKHR handle
- pSurfaceCapabilities **must** be a pointer to a VkSurfaceCapabilities2EXT 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 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 bitfield containing one bit set for each surface counter type supported.

## Valid Usage

- supportedSurfaceCounters **must** not include VK_SURFACE_COUNTER_VBLANK_EXT unless the surface queried is a display surface.

Valid values for VkSurfaceCapabilities2EXT::supportedSurfaceCounters are defined by VkSurfaceCounterFlagBitsEXT as follows:

```
typedef enum VkSurfaceCounterFlagBitsEXT {
    VK_SURFACE_COUNTER_VBLANK_EXT = 0x00000001,
} VkSurfaceCounterFlagBitsEXT;
```

- `VK_SURFACE_COUNTER_VBLANK_EXT` A counter incrementing once every time a vblank period occurs on the display associated with the surface.

VkSurfaceCapabilitiesKHR::`supportedTransforms` is a bitmask of, and VkSurfaceCapabilitiesKHR::`currentTransform` is a single bit from `VkSurfaceTransformFlagBitsKHR`, which contains the following values:

```
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,
} VkSurfaceTransformFlagBitsKHR;
```

These values are described as follows:

- `VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR`: The image content is presented without being transformed.
- `VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR`: The image content is rotated 90 degrees clockwise.
- `VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR`: The image content is rotated 180 degrees clockwise.
- `VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR`: The image content is rotated 270 degrees clockwise.
- `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR`: The image content is mirrored horizontally.
- `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR`: The image content is mirrored horizontally, then rotated 90 degrees clockwise.
- `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR`: The image content is mirrored horizontally, then rotated 180 degrees clockwise.
- `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR`: The image content is mirrored horizontally, then rotated 270 degrees clockwise.
- `VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR`: The presentation transform is not specified, and is instead determined by platform-specific considerations and mechanisms outside Vulkan.

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,
} 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.

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`. The number of format pairs supported will be greater than or equal to 1. 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.

### Valid Usage (Implicit)

- physicalDevice **must** be a valid VkPhysicalDevice handle
- surface **must** be a valid VkSurfaceKHR handle
- pSurfaceFormatCount **must** be a pointer to a uint32_t value
- If the value referenced by pSurfaceFormatCount is not 0, and pSurfaceFormats is not NULL, pSurfaceFormats **must** be a pointer to an array of pSurfaceFormatCount VkSurfaceFormatKHR 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 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*      pSurfaceInfo,
    uint32_t*                                   pSurfaceFormatCount,
    VkSurfaceFormat2KHR*                        pSurfaceFormats);
```

- physicalDevice is the physical device that will be associated with the swapchain to be created, as described for vkCreateSwapchainKHR.

- pSurfaceInfo points to an instance of the 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.

If pSurfaceFormats is NULL, then the number of format tuples supported for the given surface is returned in pSurfaceFormatCount. The number of format tuples supported will be greater than or equal to 1. 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 (Implicit)

- physicalDevice **must** be a valid VkPhysicalDevice handle
- pSurfaceInfo **must** be a pointer to a valid VkPhysicalDeviceSurfaceInfo2KHR structure
- pSurfaceFormatCount **must** be a pointer to a uint32_t value
- If the value referenced by pSurfaceFormatCount is not 0, and pSurfaceFormats is not NULL, pSurfaceFormats **must** be a 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.

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.

The VkColorSpaceKHR is defined as follows:

```
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_DCI_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,
} VkColorSpaceKHR;
```

- VK_COLOR_SPACE_SRGB_NONLINEAR_KHR: The presentation engine supports the sRGB color space.
- VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT - supports the Display-P3 color space and applies an sRGB-like OETF (defined below).
- VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT - supports the extended sRGB color space and applies a linear OETF.
- 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 OETF.
- 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.

The color components of Non-linear color space swap chain images have had the appropriate transfer function applied. Vulkan requires that all implementations support the sRGB OETF and EOTF transfer functions when using an SRGB pixel format. Other transfer functions, such as SMPTE 170M or SMPTE2084, **must** not be performed by the implementation, but **can** be performed by the application shader. This extension defines enums for VkColorSpaceKHR that correspond to the following color spaces::

*Table 34. Color Spaces and Attributes*

| Name | Red Primary | Green Primary | Blue Primary | White-point | OETF |
|---|---|---|---|---|---|
| DCI-P3 | 0.680, 0.320 | 0.265, 0.690 | 0.150, 0.060 | 0.3127, 0.3290 (D65) | Gamma 2.6 |
| 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) | 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 |
| DOLBYVISION | 0.708, 0.292 | 0.170, 0.797 | 0.131, 0.046 | 0.3127, 0.3290 (D65) | ST2084 |
| 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 |

For Opto-Electrical Transfer Function (OETF), unless otherwise specified, the values of L and E are defined as:

L - linear luminance of image $0 \leq L \leq 1$ for conventional colorimetry

E - corresponding electrical signal (value stored in memory)

### 30.5.1. sRGB OETF

$$E = \begin{cases} 1.055 \times L^{\frac{1}{2.4}} - 0.055 & \text{for } 0.0031308 \le L \le 1 \\ 12.92 \times L & \text{for } 0 \le L < 0.0031308 \end{cases}$$

### 30.5.2. Display-P3 EOTF

$$E = \begin{cases} (a \times L + b)^{2.4} & \text{for } 0.039 \le L \le 1 \\ b \times L & \text{for } 0 \le L < 0.039 \end{cases}$$

$a = 0.948$

$b = 0.052$

$c = 0.077$

### 30.5.3. Display-P3 OETF

$$E = \begin{cases} 1.055 \times L^{\frac{1}{2.4}} - 0.055 & \text{for } 0.0030186 \le L \le 1 \\ 12.92 \times L & \text{for } 0 \le L < 0.0030186 \end{cases}$$

> *Note*
>
> For most uses, the sRGB OETF is equivalent.

### 30.5.4. Extended sRGB OETF

$$E = \begin{cases} 1.055 \times L^{\frac{1}{2.4}} - 0.055 & \text{for } 0.0031308 \le L \le 7.5913 \\ 12.92 \times L & \text{for } 0 \le L < 0.0031308 \\ -f(-L) & \text{for } L < 0 \end{cases}$$

L - luminance of image is within [-0.6038, 7.5913].

E can be negative and/or > 1. That is how extended sRGB specifies colors outside the standard sRGB gamut. This means extended sRGB needs a floating point pixel format to cover the intended color range.

### 30.5.5. SMPTE 170M OETF

$$E = \begin{cases} \alpha \times L^{0.45} - (1 - \alpha) & \text{for } \beta \le L \le 1 \\ 4.5 \times L & \text{for } 0 \le L < \beta \end{cases}$$

$\alpha = 1.099 \text{and} \beta = 0.018 \text{for 10-bits andless per sample system (the values given in Rec.709)}$

$\alpha = 1.0993 \text{and} \beta = 0.0181 \text{for 12-bitsper sample system}$

### 30.5.6. SMPTE ST2084 OETF (Inverse-EOTF)

$$[E = (\frac{c_1 + c_2 \times L^{m_1}}{1 + c_3 \times L^{m_1}})^{m_2}$$

```
where:
```

$m_1 = 2610 / 4096 \times \frac{1}{4} = 0.1593017578125$
$m_2 = 2523 / 4096 \times 128 = 78.84375$
$c_1 = 3424 / 4096 = 0.8359375 = c3 - c2 + 1$
$c_2 = 2413 / 4096 \times 32 = 18.8515625$
$c_3 = 2392 / 4096 \times 32 = 18.6875$

### 30.5.7. Hybrid Log Gamma (HLG)

$$E = \begin{cases} r\sqrt{L} & \text{for } 0 \leq L \leq 1 \\ a \times \ln(L - b) + c & \text{for } 1 < L \end{cases}$$

$L$- is the signal normalized by the reference whitelevel
$r$- is the reference white level and has a signal value of$0.5$
$a = 0.17883277$and$b = 0.28466892$, and$c = 0.55991073$

### 30.5.8. AdobeRGB OETF

$E = L^{\frac{1}{2.19921875}}$

### 30.5.9. Gamma 2.6 OETF

$E = L^{\frac{1}{2.6}}$

An implementation supporting this extension indicates support for these color spaces via VkSurfaceFormatKHR structures returned from vkGetPhysicalDeviceSurfaceFormatsKHR.

Specifying the supported surface color space when calling vkCreateSwapchainKHR will create a swapchain using that color space.

Vulkan requires that all implementations support the sRGB Opto-Electrical Transfer Function (OETF) and Electro-optical transfer function (EOTF) when using an SRGB pixel format. Other transfer functions, such as SMPTE 170M, **must** not be performed by the implementation, but **can** be performed by the application shader.

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.

> *Note*
>
> If `pSurfaceFormats` includes just one entry, whose value for `format` is `VK_FORMAT_UNDEFINED`, `surface` has no preferred format. In this case, the application **can** use any valid `VkFormat` value.

*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 May 13, 2016 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.

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.

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 pointer to a `uint32_t` value
- If the value referenced by `pPresentModeCount` is not `0`, and `pPresentModes` is not `NULL`, `pPresentModes` **must** be a 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

The definition of VkPresentModeKHR is:

```
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,
} VkPresentModeKHR;
```

- VK_PRESENT_MODE_IMMEDIATE_KHR: 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: 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: 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: 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: 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: 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 35. Presentable image usage queries*

| Presentation mode | Image usage flags |
|---|---|
| VK_PRESENT_MODE_IMMEDIATE_KHR | VkSurfaceCapabilitiesKHR::supportedUsageFlags |
| VK_PRESENT_MODE_MAILBOX_KHR | VkSurfaceCapabilitiesKHR::supportedUsageFlags |
| VK_PRESENT_MODE_FIFO_KHR | VkSurfaceCapabilitiesKHR::supportedUsageFlags |
| VK_PRESENT_MODE_FIFO_RELAXED_KHR | VkSurfaceCapabilitiesKHR::supportedUsageFlags |
| VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR | VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags |
| VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR | VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags |

*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).

## 30.6. 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 vkGetDeviceGroupPresentCapabilitiesKHX(
    VkDevice                                    device,
    VkDeviceGroupPresentCapabilitiesKHX*        pDeviceGroupPresentCapabilities);
```

- `device` is the logical device.
- `pDeviceGroupPresentCapabilities` is a pointer to a structure of type VkDeviceGroupPresentCapabilitiesKHX that is filled with the logical device's capabilities.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pDeviceGroupPresentCapabilities` **must** be a pointer to a `VkDeviceGroupPresentCapabilitiesKHX` structure

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDeviceGroupPresentCapabilitiesKHX` structure is defined as:

```
typedef struct VkDeviceGroupPresentCapabilitiesKHX {
    VkStructureType                     sType;
    const void*                         pNext;
    uint32_t                            presentMask[VK_MAX_DEVICE_GROUP_SIZE_KHX];
    VkDeviceGroupPresentModeFlagsKHX    modes;
} VkDeviceGroupPresentCapabilitiesKHX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `presentMask` is an array of 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 indicating which device group presentation modes are supported. The bits returned in `modes` are:

```
typedef enum VkDeviceGroupPresentModeFlagBitsKHX {
    VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHX = 0x00000001,
    VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHX = 0x00000002,
    VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHX = 0x00000004,
    VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHX = 0x00000008,
} VkDeviceGroupPresentModeFlagBitsKHX;
```

- if `modes` contains `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHX`, any physical device with a presentation engine **can** present its own swapchain images.
- if `modes` contains `VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHX`, any physical device with a presentation engine **can** present swapchain images from any physical device in its `presentMask`.
- if `modes` contains `VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHX`, any physical device with a presentation engine **can** present the sum of swapchain images from any physical devices in its `presentMask`.
- if `modes` contains `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHX`, multiple physical devices with a presentation engine **can** each present their own swapchain images.

`modes` always has `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHX` set.

The present mode flags are also used when presenting an image, in VkDeviceGroupPresentInfoKHX::`mode`.

If a device group only includes a single physical device, then `modes` **must** equal `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHX`.

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 vkGetDeviceGroupSurfacePresentModesKHX(
    VkDevice                                    device,
    VkSurfaceKHR                                surface,
    VkDeviceGroupPresentModeFlagsKHX*           pModes);
```

- `device` is the logical device.
- `surface` is the surface.
- `pModes` is a pointer to a value of type `VkDeviceGroupPresentModeFlagsKHX` that is filled with the supported device group present modes for the surface.

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 vkGetDeviceGroupPresentCapabilitiesKHX.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `surface` **must** be a valid `VkSurfaceKHR` handle
- `pModes` **must** be a pointer to a combination of VkDeviceGroupPresentModeFlagBitsKHX values
- `pModes` **must** not be `0`

### 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`

When using `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHX`, 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 vkGetPhysicalDevicePresentRectanglesKHX(
    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 pointer to a uint32_t value
- If the value referenced by pRectCount is not 0, and pRects is not NULL, pRects **must** be a pointer to an array of pRectCount VkRect2D structures

### 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

## 30.7. 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 isn't 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 an instance of the `VkRefreshCycleDurationGOOGLE` structure.

## Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `swapchain` **must** be a valid `VkSwapchainKHR` handle
- `pDisplayTimingProperties` **must** be a pointer to a `VkRefreshCycleDurationGOOGLE` structure

## 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.

The rate at which an application renders and presents new images is known as the image present rate (IPR, a.k.a. 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 won't 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                                    device,
    VkSwapchainKHR                              swapchain,
    uint32_t*                                   pPresentationTimingCount,
    VkPastPresentationTimingGOOGLE*             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 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 pointer to a `uint32_t` value
- If the value referenced by `pPresentationTimingCount` is not `0`, and `pPresentationTimings` is not `NULL`, `pPresentationTimings` **must** be a pointer to an array of `pPresentationTimingCount` `VkPastPresentationTimingGOOGLE` structures

## Host Synchronization

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

### Return Codes

**Success**

- VK_SUCCESS

**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 VkPresentTimeInfoGOOGLE::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 VkPresentTimeInfoGOOGLE::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 VkPresentTimeInfoGOOGLE::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 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.

## 30.8. 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 swapchain at a time. Further, swapchains **cannot** be created for native windows that have a non-Vulkan graphics API surface associated with them.

The presentation engine is an abstraction for the platform's compositor or hardware/software display engine.

> *Note*
>
> 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

### 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

The possible return values for vkGetSwapchainStatusKHR() **should** be interpreted as follows:

- VK_SUCCESS indicates 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 an instance of the 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.

### Valid Usage (Implicit)

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

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 indicating parameters of swapchain creation. Bits which **can** be set include:

  ```
  typedef enum VkSwapchainCreateFlagBitsKHR {
      VK_SWAPCHAIN_CREATE_BIND_SFR_BIT_KHX = 0x00000001,
  } VkSwapchainCreateFlagBitsKHR;
  ```

  If `flags` includes `VK_SWAPCHAIN_CREATE_BIND_SFR_BIT_KHX`, then images created from the swapchain (i.e. with the `swapchain` member of VkImageSwapchainCreateInfoKHX set to this swapchain's handle) **must** use `VK_IMAGE_CREATE_BIND_SFR_BIT_KHX`.

- `surface` is the surface that the swapchain will present images to.
- `minImageCount` is the minimum number of presentable images that the application needs. The platform will either create the swapchain with at least that many images, or will fail to create the swapchain.
- `imageFormat` is a VkFormat that is valid for swapchains on the specified surface.
- `imageColorSpace` is a VkColorSpaceKHR that is valid for swapchains on the specified surface.
- `imageExtent` is the size (in pixels) of the swapchain. Behavior is platform-dependent when the image extent does not match the surface's `currentExtent` as returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`.
- `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, indicating how the application will use the swapchain's presentable images.
- `imageSharingMode` is the sharing mode used for the images of the swapchain.
- `queueFamilyIndexCount` is the number of queue families having access to the images of the swapchain in case `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`.
- `pQueueFamilyIndices` is an array of queue family indices having access to the images of the swapchain in case `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`.
- `preTransform` is a bitmask of VkSurfaceTransformFlagBitsKHR, 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 bitmask of VkCompositeAlphaFlagBitsKHR, 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` indicates whether the Vulkan implementation is allowed to discard rendering operations that affect regions of the surface which 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. Pixel shaders **may** not execute for these pixels, and thus any side affects they would have had will not occur.
  - If set to `VK_FALSE`, presentable images associated with the swapchain will own all the pixels they contain. Setting this value to `VK_TRUE` does not guarantee any clipping will occur, but allows more optimal presentation methods to be used on some platforms.

> *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 pixel shaders do not have any side effects that require them to run for all pixels in the presentable image.

- `oldSwapchain`, if not VK_NULL_HANDLE, specifies the swapchain that will be replaced by the new swapchain being created. The new swapchain will be a descendant of `oldSwapchain`. Further, any descendants of the new swapchain will also be descendants of `oldSwapchain`. Upon calling `vkCreateSwapchainKHR` with a `oldSwapchain` that is not VK_NULL_HANDLE, any images not acquired by the application **may** be freed by the implementation, which **may** occur even if creation of the new swapchain fails. The application **must** destroy the old swapchain to free all memory associated with the old swapchain. The application **must** wait for the completion of any outstanding rendering to images it currently has acquired at the time the swapchain is destroyed. The application **can** continue to present any images it acquired and has not yet presented using the old swapchain, as long as it has not entered a state that causes it to return `VK_ERROR_OUT_OF_DATE_KHR`. However, the

application **cannot** acquire any more images from the old swapchain regardless of whether or not creation of the new swapchain succeeds. 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
- `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
- `imageUsage` **must** be a subset of the supported usage flags present in the `supportedUsageFlags` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface if `presentMode` is set to `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 `sharedPresentSupportedUsageFlags` member of the `VkSharedPresentSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilities2KHR` for the surface if `presentMode` is set to either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`
- If `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` **must** be a pointer to an array of `queueFamilyIndexCount` `uint32_t` values
- If `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` **must** be greater than `1`
- `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

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR
- pNext **must** be NULL or a pointer to a valid instance of VkDeviceGroupSwapchainCreateInfoKHX
- 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

If the pNext list of VkSwapchainCreateInfoKHR includes a VkDeviceGroupSwapchainCreateInfoKHX structure, then that structure includes a set of device group present modes that the swapchain **can** be used with.

The VkDeviceGroupSwapchainCreateInfoKHX structure is defined as:

```
typedef struct VkDeviceGroupSwapchainCreateInfoKHX {
    VkStructureType                     sType;
    const void*                         pNext;
    VkDeviceGroupPresentModeFlagsKHX    modes;
} VkDeviceGroupSwapchainCreateInfoKHX;
```

- 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_KHX.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHX`
- `modes` **must** be a valid combination of VkDeviceGroupPresentModeFlagBitsKHX values
- `modes` **must** not be `0`

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 containing a bit set for each surface counter 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`
- `pNext` **must** be `NULL`
- `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 pointer to a uint64_t value

### Return Codes

**Success**

- VK_SUCCESS
- VK_ERROR_DEVICE_LOST
- VK_ERROR_OUT_OF_DATE_KHR

As mentioned above, if vkCreateSwapchainKHR succeeds, it will return a handle to a swapchain that contains 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 | VK_IMAGE_CREATE_BIND_SFR_BIT_KHX if VK_SWAPCHAIN_CREATE_BIND_SFR_BIT_KHX is set, else 0 |
| imageType | VK_IMAGE_TYPE_2D |
| format | pCreateInfo→imageFormat |
| extent | {pCreateInfo→imageExtent.width, pCreateInfo→imageExtent.height, 1} |
| mipLevels | 1 |
| arrayLayers | pCreateInfo→imageArrayLayers |

| VkImageCreateInfo Field | Value |
|---|---|
| samples | VK_SAMPLE_COUNT_1_BIT |
| tiling | VK_IMAGE_TILING_OPTIMAL |
| usage | pCreateInfo→imageUsage |
| sharingMode | pCreateInfo→imageSharingMode |
| queueFamilyIndexCount | pCreateInfo→queueFamilyIndexCount |
| pQueueFamilyIndices | pCreateInfo→pQueueFamilyIndices |
| initialLayout | VK_IMAGE_LAYOUT_UNDEFINED |

The VkSurfaceKHR associated with a swapchain **must** not be destroyed until after the swapchain is destroyed.

The native window referred to by surface **must** not already be associated with a swapchain other than oldSwapchain, or with a non-Vulkan graphics API surface.

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).

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 platform. 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.

If a swapchain associated with a display surface is destroyed and there are no valid descendants of that swapchain, the implementation **must** either revert any display resources modified by presenting images with the swapchain to their state prior to the first present performed with the swapchain and its ancestors, or leave such resources in their current state.

### 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 pointer to a valid `VkAllocationCallbacks` structure

### 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.

vkCreateSharedSwapchains 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 pointer to an array of swapchainCount valid VkSwapchainCreateInfoKHR structures
- If pAllocator is not NULL, pAllocator **must** be a pointer to a valid VkAllocationCallbacks structure
- pSwapchains **must** be a 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 pointer to a `uint32_t` value
- If the value referenced by `pSwapchainImageCount` is not `0`, and `pSwapchainImages` is not `NULL`, `pSwapchainImages` **must** be a pointer to an array of `pSwapchainImageCount` `VkImage` handles

## 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.

The implementation will have already allocated and bound the memory backing the `VkImages` returned by `vkGetSwapchainImagesKHR`. The memory for each image will not alias with the memory for other images or with any `VkDeviceMemory` object. As such, performing any operation affecting the binding of memory to a presentable image results in undefined behavior. 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 destroying a presentable image with vkDestroyImage results in undefined behavior. See vkDestroySwapchainKHR for further details on the lifetime of presentable images.

Images **can** also be created by using vkCreateImage with VkImageSwapchainCreateInfoKHX and bound to swapchain memory using vkBindImageMemory2KHX with VkBindImageMemorySwapchainInfoKHX. 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 swapchain from which an image is being acquired.
- `timeout` indicates 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` that is set to the index of the next image to use (i.e. an index into the array of images returned by `vkGetSwapchainImagesKHR`).

## Valid Usage

- `swapchain` **must** not have been replaced by being passed as the `VkSwapchainCreateInfoKHR::oldSwapchain` value to `vkCreateSwapchainKHR`
- If `semaphore` is not VK_NULL_HANDLE it **must** be unsignaled
- 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

## 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 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`

## 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`

When successful, vkAcquireNextImageKHR acquires a presentable image that the application **can** use, and sets pImageIndex to the index of that image. 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.

As mentioned above, the presentation engine controls the order in which presentable images are made available to the application. This allows the platform to handle special situations. The order in which images are acquired is implementation-dependent. Images **may** be acquired in a seemingly random order that is not a simple round-robin.

If a swapchain has enough presentable images, applications **can** acquire multiple images without an intervening vkQueuePresentKHR. Applications **can** present images in a different order than the order in which they were acquired.

If timeout is 0, vkAcquireNextImageKHR will not block, but will either succeed or return VK_NOT_READY. If timeout is UINT64_MAX, the function will not return until an image is acquired from the presentation engine. Other values for timeout will cause the function to return when an image becomes available, or when the specified number of nanoseconds have passed (in which case it will return VK_TIMEOUT). An error **can** also cause vkAcquireNextImageKHR to return early.

> *Note*
>
> As mentioned above, the presentation engine **may** be asynchronous with respect to the application and/or logical device. vkAcquireNextImageKHR **may** return as soon as it **can** identify which image will be acquired, and **can** guarantee that semaphore and fence will be signaled by the presentation engine; and **may** not successfully return sooner. The application uses timeout to specify how long vkAcquireNextImageKHR waits for an image to become acquired.

Applications **cannot** rely on vkAcquireNextImageKHR blocking in order to meter their rendering speed. Various factors **can** interrupt vkAcquireNextImageKHR from blocking.

> *Note*
>
> For example, if an error occurs, vkAcquireNextImageKHR **may** return even though no image is available. As another example, some presentation engines are able to enqueue an unbounded number of presentation and acquire next image operations such that vkAcquireNextImageKHR never needs to wait for completion of outstanding present operations before returning.

The availability of presentable images is influenced by factors such as the implementation of the presentation engine, the VkPresentModeKHR being used, the number of images in the swapchain, the number of images that the application has acquired at any given time, and the performance of the application. The value of VkSurfaceCapabilitiesKHR::minImageCount indicates how many images **must** be in the swapchain in order for vkAcquireNextImageKHR to acquire an image if the application currently has no acquired images.

Let $n$ be the total number of images in the swapchain, $m$ be the value of VkSurfaceCapabilitiesKHR::minImageCount, and $a$ be the number of presentable images that the application has currently acquired

(i.e. images acquired with vkAcquireNextImageKHR, but not yet presented with vkQueuePresentKHR). vkAcquireNextImageKHR **can** always succeed if a ≤ n - m at the time vkAcquireNextImageKHR is called. vkAcquireNextImageKHR **should** not be called if a > n - m with a timeout of UINT64_MAX; in such a case, vkAcquireNextImageKHR **may** block indefinitely.

> *Note*
>
> For example, if the minImageCount member of VkSurfaceCapabilitiesKHR is 2, and the application creates a swapchain with 2 presentable images, the application **can** acquire one image, and **must** present it before trying to acquire another image.
>
> If we modify this example so that the application wishes to acquire up to 3 presentable images simultaneously, it **must** request a minimum image count of 4 when creating the swapchain.

If semaphore is not VK_NULL_HANDLE, the semaphore **must** be unsignaled and not have any uncompleted signal or wait operations pending. It will become signaled when the application **can** use the image. Queue operations that access the image contents **must** wait until the semaphore signals; typically applications **should** include the semaphore in the pWaitSemaphores list for the queue submission that transitions the image away from the VK_IMAGE_LAYOUT_PRESENT_SRC_KHR layout. Use of the 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 and not have any uncompleted signal operations pending. It will become signaled when the application **can** use the image. Applications **can** use this to meter their frame generation work to match the presentation rate.

semaphore and fence **must** not both be equal to VK_NULL_HANDLE. An application **must** wait until either the semaphore or fence is signaled before using the presentable image.

semaphore and fence **may** already be signaled when vkAcquireNextImageKHR returns, if the image is being acquired for the first time, or if the presentable image is immediately ready for use.

A successful call to vkAcquireNextImageKHR counts as a signal operation on semaphore for the purposes of queue forward-progress requirements. The semaphore is guaranteed to signal, so a wait operation **can** be queued for the semaphore without risk of deadlock.

The vkCmdWaitEvents or vkCmdPipelineBarrier used to transition the image away from VK_IMAGE_LAYOUT_PRESENT_SRC_KHR layout **must** have dstStageMask and dstAccessMask parameters set based on the next use of the image. The application **must** use implicit ordering guarantees and execution dependencies to prevent the image transition from occurring before the semaphore passed to vkAcquireNextImageKHR has signaled.

> *Note*
>
> When the presentable image will be written by some stage S, the recommended idiom for ensuring the semaphore signals before the transition occurs is:
>
> - The batch that contains the transition includes the image-acquire semaphore in the list of semaphores to wait for, with a wait stage mask that includes S.
> - The pipeline barrier that performs the transition includes S in both the `srcStageMask` and `dstStageMask`.
>
> This causes the pipeline barrier to wait at S until the semaphore signals before performing the transition and memory barrier, while allowing earlier pipeline stages of subsequent commands to proceed.

After a successful return, the image indicated by `pImageIndex` will still be in the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout if it was previously presented, or in the `VK_IMAGE_LAYOUT_UNDEFINED` layout if this is the first time it has been acquired.

> *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 applications to decide whether they prefer 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, parts of the native surface **may** be undefined, parts of the presented image **may** have been clipped before presentation, and/or the image **may** have been scaled (uniformly or not uniformly) before presentation. 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.

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`.

To acquire an available presentable image to use, and retrieve the index of that image, call:

```
VkResult vkAcquireNextImage2KHX(
    VkDevice                                    device,
    const VkAcquireNextImageInfoKHX*            pAcquireInfo,
    uint32_t*                                   pImageIndex);
```

- `device` is the device associated with `swapchain`.
- `pAcquireInfo` is a pointer to a structure of type VkAcquireNextImageInfoKHX 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 (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pAcquireInfo` **must** be a pointer to a valid `VkAcquireNextImageInfoKHX` structure
- `pImageIndex` **must** be a 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

The VkAcquireNextImageInfoKHX structure is defined as:

```
typedef struct VkAcquireNextImageInfoKHX {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
    uint64_t           timeout;
    VkSemaphore        semaphore;
    VkFence            fence;
    uint32_t           deviceMask;
} VkAcquireNextImageInfoKHX;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to an extension-specific structure.
- swapchain is the swapchain from which an image is being acquired.
- timeout indicates 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*

vkAcquireNextImage2KHX 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

- If semaphore is not VK_NULL_HANDLE it **must** be unsignaled
- 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
- deviceMask **must** be a valid device mask
- deviceMask **must** not be zero

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHX
- 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

### 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 an instance of the VkPresentInfoKHR structure specifying the parameters of the presentation.

## Valid Usage

- Any given 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 unsignal operation defined by the elements of the pWaitSemaphores member of pPresentInfo executes on queue, no other queue **must** be 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.

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.

## Valid Usage (Implicit)

- queue **must** be a valid VkQueue handle
- pPresentInfo **must** be a 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

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, if not NULL, is 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 an array of VkSwapchainKHR objects with swapchainCount entries. A given swapchain **must** not appear in this list more than once.
- pImageIndices is 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 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.

### Valid Usage

- Any given 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

### 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 VkDisplayPresentInfoKHR, VkPresentRegionsKHR, VkDeviceGroupPresentInfoKHX, or VkPresentTimesInfoGOOGLE
- Each sType member in the pNext chain **must** be unique
- If waitSemaphoreCount is not 0, and pWaitSemaphores is not NULL, pWaitSemaphores **must** be a pointer to an array of waitSemaphoreCount valid VkSemaphore handles
- pSwapchains **must** be a pointer to an array of swapchainCount valid VkSwapchainKHR handles
- pImageIndices **must** be a pointer to an array of swapchainCount uint32_t values
- If pResults is not NULL, pResults **must** be a pointer to an array of swapchainCount VkResult values
- swapchainCount **must** be greater than 0

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 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 pointer to an array of rectangleCount 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` list of VkPresentInfoKHR includes a `VkDeviceGroupPresentInfoKHX` structure, then that structure includes an array of device masks and a device group present mode.

The `VkDeviceGroupPresentInfoKHX` structure is defined as:

```
typedef struct VkDeviceGroupPresentInfoKHX {
    VkStructureType                         sType;
    const void*                             pNext;
    uint32_t                                swapchainCount;
    const uint32_t*                         pDeviceMasks;
    VkDeviceGroupPresentModeFlagBitsKHX     mode;
} VkDeviceGroupPresentInfoKHX;
```

- `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 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_KHX, 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 VkDeviceGroupPresentCapabilitiesKHX.

If mode is VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHX, 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 VkDeviceGroupPresentCapabilitiesKHX::presentMask.

If mode is VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHX, 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 VkDeviceGroupPresentCapabilitiesKHX::presentMask.

If mode is VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHX, 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 VkDeviceGroupPresentCapabilitiesKHX.

If VkDeviceGroupPresentInfoKHX is not provided or swapchainCount is zero then the masks are considered to be 1. If VkDeviceGroupPresentInfoKHX is not provided, mode is considered to be VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHX.

## Valid Usage

- swapchainCount **must** equal 0 or VkPresentInfoKHR::swapchainCount
- If mode is VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHX, then each element of pDeviceMasks **must** have exactly one bit set, and the corresponding element of VkDeviceGroupPresentCapabilitiesKHX::presentMask **must** be non-zero
- If mode is VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHX, 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 VkDeviceGroupPresentCapabilitiesKHX::presentMask.
- If mode is VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHX, then each element of pDeviceMasks **must** have a value for which all set bits are set in one of the elements of VkDeviceGroupPresentCapabilitiesKHX::presentMask
- If mode is VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHX, then for each bit set in each element of pDeviceMasks, the corresponding element of VkDeviceGroupPresentCapabilitiesKHX::presentMask **must** be non-zero
- The value of each element of pDeviceMasks **must** be equal to the device mask passed in VkAcquireNextImageInfoKHX::deviceMask when the image index was last acquired
- mode **must** have exactly one bit set, and that bit **must** have been included in VkDeviceGroupSwapchainCreateInfoKHX::modes

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHX`
- If `swapchainCount` is not `0`, `pDeviceMasks` **must** be a pointer to an array of `swapchainCount` `uint32_t` values
- `mode` **must** be a valid VkDeviceGroupPresentModeFlagBitsKHX 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 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 indicates 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 indicates that the presentation engine **may** display the image at any time. This is useful when the application desires to provide presentID, but doesn't need a specific desiredPresentTime.

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_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.

## 30.9. 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 the array of swapchainCount VkSwapchainKHR handles.
- pMetadata is a pointer to the array of swapchainCount VkHdrMetadataEXT structures.

### Valid Usage (Implicit)

- device **must** be a valid VkDevice handle
- pSwapchains **must** be a pointer to an array of swapchainCount valid VkSwapchainKHR handles
- pMetadata **must** be a pointer to an array of swapchainCount valid VkHdrMetadataEXT structures
- swapchainCount **must** be greater than 0

```
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

*note*

The validity and use of this data is outside the scope of Vulkan and thus no Valid Usage is given.

# Chapter 31. Extended Functionality

Additional functionality **may** be provided by layers or extensions. A layer **cannot** add or modify Vulkan commands, while an extension **may** do so.

The set of layers to enable is specified when creating an instance, and those layers are able to intercept any Vulkan command dispatched to that instance or any of its child objects.

Extensions can operate at either the instance or device *extension scope*. Enabled instance extensions are able to affect the operation of the instance and any of its child objects, while device extensions **may** only be available on a subset of physical devices, **must** be individually enabled per-device, and only affect the operation of the devices where they are enabled.

Examples of these might be:

- Whole API validation is an example of a layer.
- Debug capabilities might make a good instance extension.
- A layer that provides hardware-specific performance telemetry and analysis could be a layer that is only active for devices created from compatible physical devices.
- Functions to allow an application to use additional hardware features beyond the core would be a good candidate for a device extension.

## 31.1. Layers

When a layer is enabled, it inserts itself into the call chain for Vulkan commands the layer is interested in. A common use of layers is to validate application behavior during development. For example, the implementation will not check that Vulkan 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.

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*                                   pPropertyCount,
    VkLayerProperties*                          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 pointer to a uint32_t value
- If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties **must** be a 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 a null-terminated UTF-8 string specifying 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 the API Version Numbers and Semantics section.

- `implementationVersion` is the version of this layer. It is an integer, increasing with backward compatible changes.
- `description` is a null-terminated UTF-8 string providing 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.

### 31.1.1. 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 pointer to a uint32_t value
- If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties **must** be a 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.

## 31.2. 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.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.

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.

### Valid Usage (Implicit)

- If `pLayerName` is not `NULL`, `pLayerName` **must** be a null-terminated string
- `pPropertyCount` **must** be a pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a 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.

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.

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.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- If `pLayerName` is not `NULL`, `pLayerName` **must** be a null-terminated string
- `pPropertyCount` **must** be a pointer to a `uint32_t` value
- If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` **must** be a 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 a null-terminated string specifying the name of the extension.
- `specVersion` is the version of this extension. It is an integer, incremented with backward compatible changes.

## 31.2.1. Instance Extensions and Device Extensions

This section provides some guidelines and rules for when to expose new functionality as an instance extension, as a device extension, or as both. The decision depends on the scope of the new functionality; such as whether it extends instance-level or device-level functionality. All Vulkan commands, structures, and enumerants are considered either instance-level, physical-device-level, or device-level.

Commands that are dispatched from instances (`VkInstance`) are considered instance-level commands. Any structure, enumerated type, and enumerant that is used with instance-level commands are considered instance-level objects. New instance-level extension functionality **must** be structured within an instance extension.

Any command or object that **must** be used after calling vkCreateDevice is a device-level command or

object. These objects include all children of `VkDevice` objects, such as queues (`VkQueue`) and command buffers (`VkCommandBuffer`). New device-level extension functionality **may** be structured within a device extension.

Commands that are dispatched from physical devices (`VkPhysicalDevice`) are considered physical-device-level commands. Any structure, enumerated type, and enumerant that is used with physical-device-level commands, and not used with instance-level commands, are considered physical-device-level objects. Vulkan 1.0 requires all new physical-device-level extension functionality to be structured within an instance extension. In order to avoid using an instance extension, which often requires loader support, the VK_KHR_get_physical_device_properties2 extension allows physical-device-level extension functionality to be implemented within device extensions (which **must** depend on the VK_KHR_get_physical_device_properties2 extension).

# Chapter 32. Features, Limits, and Formats

Vulkan is designed to support a wide range of hardware and as such there are a number of features, limits, and formats which are not supported on all hardware. Features describe functionality that is not **required** and which **must** be explicitly enabled. Limits describe implementation-dependent minimums, maximums, and other device characteristics that an application **may** need to be aware of. 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.

> *Note on extensibility*
>
> The features and limits are reported via basic structures (that is VkPhysicalDeviceFeatures and VkPhysicalDeviceLimits), as well as extensible structures (VkPhysicalDeviceFeatures2KHR and VkPhysicalDeviceProperties2KHR) which were added in VK_KHR_get_physical_device_properties2. When new features or limits are added in future Vulkan version or extensions, each extension **should** introduce one new feature structure and/or limit structure (as needed). These structures **can** be added to the pNext chain of the VkPhysicalDeviceFeatures2KHR and VkPhysicalDeviceProperties2KHR structures, respectively.

## 32.1. Features

The Specification defines a set of fine-grained features that are not **required**, but **may** be supported by a Vulkan implementation. Support for features is reported and enabled on a per-feature basis. Features are properties of the physical device.

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 indicates that the feature is supported on this physical device, and VK_FALSE indicates that the feature is not supported.

**Valid Usage (Implicit)**

- physicalDevice **must** be a valid VkPhysicalDevice handle
- pFeatures **must** be a 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. If an application uses a

feature without enabling it at VkDevice creation time, the device behavior is undefined. The validation layer will warn if features are used without being enabled.

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 VkPhysicalDeviceFeatures2KHR 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 VkPhysicalDeviceFeatures2KHR 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 vkGetPhysicalDeviceFeatures2KHR(
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceFeatures2KHR*               pFeatures);
```

- physicalDevice is the physical device from which to query the supported features.
- pFeatures is a pointer to a VkPhysicalDeviceFeatures2KHR structure in which the physical device features are returned.

Each structure in pFeatures and its pNext chain contain members corresponding to fine-grained features. vkGetPhysicalDeviceFeatures2KHR 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 pointer to a VkPhysicalDeviceFeatures2KHR structure

The VkPhysicalDeviceFeatures2KHR structure is defined as:

```
typedef struct VkPhysicalDeviceFeatures2KHR {
    VkStructureType             sType;
    void*                       pNext;
    VkPhysicalDeviceFeatures    features;
} VkPhysicalDeviceFeatures2KHR;
```

The `VkPhysicalDeviceFeatures2KHR` 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 structure of type VkPhysicalDeviceFeatures 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 vkGetPhysicalDeviceFeatures2KHR 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_KHR`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkPhysicalDeviceMultiviewFeaturesKHX

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` indicates 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.

> *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 in a given SPIR-V block is determined to be out of bounds, then any other access of the same type (load, store, or atomic) in the same SPIR-V block 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 address of the attribute plus the size of the attribute is greater than the size of the bound buffer. 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, out of bounds accesses **may** corrupt any memory within the process and cause undefined behavior up to and including application termination.

- `fullDrawIndexUint32` indicates 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` indicates 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 indicates 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 indicates 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 indicates whether shader modules **can** declare the Geometry capability.
- tessellationShader indicates 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 indicates whether shader modules **can** declare the Tessellation capability.
- sampleRateShading indicates whether per-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 indicates whether shader modules **can** declare the SampleRateShading capability.
- dualSrcBlend indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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. Similarly, the viewportCount parameter to the vkCmdSetViewport command and the scissorCount parameter to the vkCmdSetScissor command **must** be 1, and the firstViewport parameter to the vkCmdSetViewport command and the firstScissor parameter to the vkCmdSetScissor command **must** be 0.
- samplerAnisotropy indicates whether anisotropic filtering is supported. If this feature is not enabled, the maxAnisotropy member of the VkSamplerCreateInfo structure **must** be 1.0.
- textureCompressionETC2 indicates 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

vkGetPhysicalDeviceFormatProperties and vkGetPhysicalDeviceImageFormatProperties **can** be used to check for additional supported properties of individual formats.

- textureCompressionASTC_LDR indicates 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

  vkGetPhysicalDeviceFormatProperties and vkGetPhysicalDeviceImageFormatProperties **can** be used to check for additional supported properties of individual formats.

- textureCompressionBC indicates 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

vkGetPhysicalDeviceFormatProperties and vkGetPhysicalDeviceImageFormatProperties **can** be used to check for additional supported properties of individual formats.

- occlusionQueryPrecise indicates 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 indicates 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 indicates 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 NonWriteable decoration (or the readonly memory qualifier in GLSL).
- fragmentStoresAndAtomics indicates 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 NonWriteable decoration (or the readonly memory qualifier in GLSL).
- shaderTessellationAndGeometryPointSize indicates 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 indicates 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` indicates 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 indicates whether shader modules **can** declare the `ImageGatherExtended` capability.
- `shaderStorageImageExtendedFormats` indicates whether the extended storage image formats are available in shader code. If this feature is not enabled, the formats requiring the `StorageImageExtendedFormats` capability are not supported for storage images. This also indicates whether shader modules **can** declare the `StorageImageExtendedFormats` capability.
- `shaderStorageImageMultisample` indicates 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 indicates whether shader modules **can** declare the `StorageImageMultisample` capability.
- `shaderStorageImageReadWithoutFormat` indicates 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 indicates whether shader modules **can** declare the `StorageImageReadWithoutFormat` capability.
- `shaderStorageImageWriteWithoutFormat` indicates 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 indicates whether shader modules **can** declare the `StorageImageWriteWithoutFormat` capability.
- `shaderUniformBufferArrayDynamicIndexing` indicates 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 indicates whether shader modules **can** declare the `UniformBufferArrayDynamicIndexing` capability.
- `shaderSampledImageArrayDynamicIndexing` indicates 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 indicates whether shader modules **can** declare the `SampledImageArrayDynamicIndexing` capability.
- `shaderStorageBufferArrayDynamicIndexing` indicates 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 indicates whether shader modules **can** declare the `StorageBufferArrayDynamicIndexing` capability.
- `shaderStorageImageArrayDynamicIndexing` indicates 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 indicates whether shader modules **can** declare the StorageImageArrayDynamicIndexing capability.

- shaderClipDistance indicates 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 indicates whether shader modules **can** declare the ClipDistance capability.
- shaderCullDistance indicates 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 indicates whether shader modules **can** declare the CullDistance capability.
- shaderFloat64 indicates 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 indicates whether shader modules **can** declare the Float64 capability.
- shaderInt64 indicates 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 indicates whether shader modules **can** declare the Int64 capability.
- shaderInt16 indicates 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 indicates whether shader modules **can** declare the Int16 capability.
- shaderResourceResidency indicates 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 indicates whether shader modules **can** declare the SparseResidency capability. The feature requires at least one of the sparseResidency* features to be supported.
- shaderResourceMinLod indicates whether image operations that specify the minimum resource level-of-detail (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 indicates whether shader modules **can** declare the MinLod capability.
- sparseBinding indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates 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 indicates whether a secondary command buffer **may** be executed while a query is active.

### Valid Usage

- If any member of this structure is VK_FALSE, as returned by vkGetPhysicalDeviceFeatures, then it **must** be VK_FALSE when passed as part of the VkDeviceCreateInfo struct when creating a device

The VkPhysicalDeviceMultiviewFeaturesKHX structure is defined as:

```
typedef struct VkPhysicalDeviceMultiviewFeaturesKHX {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           multiview;
    VkBool32           multiviewGeometryShader;
    VkBool32           multiviewTessellationShader;
} VkPhysicalDeviceMultiviewFeaturesKHX;
```

The members of the `VkPhysicalDeviceMultiviewFeaturesKHX` structure describe the following features:

- `multiview` indicates 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` indicates 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` indicates 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 `VkPhysicalDeviceMultiviewFeaturesKHX` structure is included in the `pNext` chain of vkGetPhysicalDeviceFeatures2KHR, it is filled with values indicating whether each feature is supported. `VkPhysicalDeviceMultiviewFeaturesKHX` **can** also be used in the `pNext` chain of vkCreateDevice to enable 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_KHX`

### 32.1.1. Feature Requirements

All Vulkan graphics implementations **must** support the following features:

- `robustBufferAccess`.

All other features are not **required** by the Specification.

## 32.2. Limits

There are a variety of implementation-dependent limits.

The VkPhysicalDeviceLimits are properties of the physical device. These are available in the limits member of the VkPhysicalDeviceProperties structure which is returned from vkGetPhysicalDeviceProperties.

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;
```

- `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. 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. 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. 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. 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. 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. 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. 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 descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptors with a type of VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER count against this limit. See Sampler and Combined Image Sampler.
- maxDescriptorSetUniformBuffers is the maximum number of uniform buffers that **can** be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC 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 descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. See Dynamic Uniform Buffer.
- maxDescriptorSetStorageBuffers is the maximum number of storage buffers that **can** be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC 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 descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. See Dynamic Storage Buffer.
- maxDescriptorSetSampledImages is the maximum number of sampled images that **can** be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. 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. See Combined Image Sampler, Sampled Image, and Uniform Texel Buffer.
- maxDescriptorSetStorageImages is the maximum number of storage images that **can** be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER count against this limit. See Storage Image, and Storage Texel Buffer.
- maxDescriptorSetInputAttachments is the maximum number of input attachments that **can** be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptors with a type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT 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, of all variables declared with the `WorkgroupLocal` storage class in shader modules (or with the `shared` storage qualifier in GLSL) in the compute shader stage.
- `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 and 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 specified by the `LocalSize` execution mode and 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. $2^{\texttt{subTexelPrecisionBits}}$ is the actual number of divisions along each axis of the image represented. The filtering hardware will snap to these locations when computing 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. $2^{\texttt{mipmapPrecisionBits}}$ is the actual number of divisions.

> *Note*
>
> For example, if this value is 2 bits then when linearly filtering between two levels, each level could: contribute: 0%, 33%, 66%, or 100% (this is just an example and the amount of contribution **should** be covered by different equations in the spec).

- `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 level of detail 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 [-`maxSamplerLodBias`,+`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 × `size`, 2 × `size` - 1], where `size` = max(`maxViewportDimensions`[0], `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 [-`size` + 1, 2 × `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. When a buffer view is created for a buffer which was created with VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT set in the usage member of the VkBufferCreateInfo structure, the offset **must** be an integer multiple of this limit.

- 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 or ConstOffsets image operands of any of the OpImage*Gather image instructions.
- maxTexelGatherOffset is the maximum offset value for the Offset 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[1] of VkSampleCountFlagBits bits indicating the color sample counts that are supported for all framebuffer color attachments.
- framebufferDepthSampleCounts is a bitmask[1] of VkSampleCountFlagBits bits indicating the supported depth sample counts for all framebuffer depth/stencil attachments, when the format includes a depth component.
- framebufferStencilSampleCounts is a bitmask[1] of VkSampleCountFlagBits bits indicating the supported stencil sample counts for all framebuffer depth/stencil attachments, when the format includes a stencil component.

- framebufferNoAttachmentsSampleCounts is a bitmask[1] of VkSampleCountFlagBits bits 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[1] of VkSampleCountFlagBits bits 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[1] of VkSampleCountFlagBits bits 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[1] of VkSampleCountFlagBits bits 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[1] of VkSampleCountFlagBits bits 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[1] of VkSampleCountFlagBits bits 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 indicates 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` indicates 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` indicates 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.
- `maxDiscardRectangles` is the maximum number of active discard rectangles. `maxDiscardRectangles` is a member of the VkPhysicalDeviceDiscardRectanglePropertiesEXT structure. This limit can be queried by setting the `pNext` pointer from a VkPhysicalDeviceProperties2KHR object to an instance of VkPhysicalDeviceDiscardRectanglePropertiesEXT and using `vkGetPhysicalDeviceProperties2KHR` to fill out the members.

**1**

For all bitmasks of type VkSampleCountFlags above, possible values include:

```
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,
} 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.

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:

- `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.

### Valid Usage (Implicit)

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

The `VkPhysicalDeviceMultiviewPropertiesKHX` structure is defined as:

```
typedef struct VkPhysicalDeviceMultiviewPropertiesKHX {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxMultiviewViewCount;
    uint32_t           maxMultiviewInstanceIndex;
} VkPhysicalDeviceMultiviewPropertiesKHX;
```

The members of the `VkPhysicalDeviceMultiviewPropertiesKHX` structure describe the following implementation-dependent limits:

- 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 VkPhysicalDeviceMultiviewPropertiesKHX structure is included in the pNext chain of VkPhysicalDeviceProperties2KHR, it is filled with the implementation-dependent limits.

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:

- maxDiscardRectangles is the maximum number of discard rectangles that can be specified.

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT
- pNext **must** be NULL

If the VkPhysicalDeviceDiscardRectanglePropertiesEXT structure is included in the pNext chain of VkPhysicalDeviceProperties2KHR, it is filled with the implementation-dependent limits.

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:

- 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 VkPhysicalDeviceProperties2KHR, it is filled with the implementation-dependent limits.

### 32.2.1. 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 36. Required Limit Types*

| Type | Limit | Feature |
|---|---|---|
| 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 | 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 | - |

| Type | Limit | Feature |
|---|---|---|
| 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 | - |
| 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 | - |

| Type | Limit | Feature |
|---|---|---|
| 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 | - |
| 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 |

| Type | Limit | Feature |
|---|---|---|
| 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 |

*Table 37. Required Limits*

| Limit | Unsupported Limit | Supported Limit | Limit Type[1] |
|---|---|---|---|
| `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 [2] | min |
| `maxDescriptorSetSamplers` | - | 96 [8] | min, $n$ × PerStage |

| Limit | Unsupported Limit | Supported Limit | Limit Type[1] |
|---|---|---|---|
| maxDescriptorSetUniformBuffers | - | 72 [8] | min, $n$ × PerStage |
| maxDescriptorSetUniformBuffersDynamic | - | 8 | min |
| maxDescriptorSetStorageBuffers | - | 24 [8] | min, $n$ × PerStage |
| maxDescriptorSetStorageBuffersDynamic | - | 4 | min |
| maxDescriptorSetSampledImages | - | 96 [8] | min, $n$ × PerStage |
| maxDescriptorSetStorageImages | - | 24 [8] | min, $n$ × PerStage |
| maxDescriptorSetInputAttachments | - | 4 | min |
| maxVertexInputAttributes | - | 16 | min |
| maxVertexInputBindings | - | 16 | min |
| 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 |

| Limit | Unsupported Limit | Supported Limit | Limit Type[1] |
|---|---|---|---|
| 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) [3] | min |
| viewportBoundsRange | - | (-8192,8191) [4] | (max,min) |
| 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 [5] | max |
| maxInterpolationOffset | 0.0 | 0.5 - (1 ULP) [5] | min |
| subPixelInterpolationOffsetBits | 0 | 4 [5] | 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 |

| Limit | Unsupported Limit | Supported Limit | Limit Type[1] |
|---|---|---|---|
| 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 |
| 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)[6] | (max,min) |
| lineWidthRange | (1.0,1.0) | (1.0,8.0 - ULP)[7] | (max,min) |
| pointSizeGranularity | 0.0 | 1.0 [6] | max, fixed point increment |
| lineWidthGranularity | 0.0 | 1.0 [7] | max, fixed point increment |
| strictLines | - | - | implementation dependent |
| standardSampleLocations | - | - | implementation dependent |

| Limit | Unsupported Limit | Supported Limit | Limit Type[1] |
|---|---|---|---|
| optimalBufferCopyOffsetAlignment | - | - | recommendation |
| optimalBufferCopyRowPitchAlignment | - | - | recommendation |
| nonCoherentAtomSize | - | 256 | max |
| maxPushDescriptors | - | 32 | min |
| maxMultiviewViewCount | - | 6 | min |
| maxMultiviewInstanceIndex | - | $2^{27}$-1 | min |
| maxDiscardRectangles | 0 | 4 | min |
| perViewPositionAllComponents | - | - | implementation dependent |

**1**

The **Limit Type** column indicates 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: [minInterpolationOffset, 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 `maxDescriptorSet*` limit is *n* times the corresponding `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).

# 32.3. Formats

The features for the set of formats (VkFormat) supported by the implementation are queried individually using the vkGetPhysicalDeviceFormatProperties command.

## 32.3.1. Format Definition

The available formats are defined by the VkFormat enumeration:

```
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_SSCALED = 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_SSCALED = 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_SSCALED = 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_SSCALED_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_SSCALED_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_SSCALED_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_SSCALED = 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_SSCALED = 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_SSCALED = 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_SSCALED = 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_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,
} VkFormat;
```

`VK_FORMAT_UNDEFINED`

The format is not specified.

`VK_FORMAT_R4G4_UNORM_PACK8`

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`

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`

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`

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`

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`

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`

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`

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`

A one-component, 8-bit unsigned normalized format that has a single 8-bit R component.

`VK_FORMAT_R8_SNORM`

A one-component, 8-bit signed normalized format that has a single 8-bit R component.

`VK_FORMAT_R8_USCALED`

A one-component, 8-bit unsigned scaled integer format that has a single 8-bit R component.

`VK_FORMAT_R8_SSCALED`

A one-component, 8-bit signed scaled integer format that has a single 8-bit R component.

`VK_FORMAT_R8_UINT`

A one-component, 8-bit unsigned integer format that has a single 8-bit R component.

`VK_FORMAT_R8_SINT`

A one-component, 8-bit signed integer format that has a single 8-bit R component.

`VK_FORMAT_R8_SRGB`

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`

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`

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`

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_SSCALED`

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`

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`

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`

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`

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`

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`

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_SSCALED`

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`

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`

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`

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`

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`

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`

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_SSCALED`

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`

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`

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`

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`

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`

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`

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_SSCALED`

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`

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`

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`

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`

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`

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`

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_SSCALED`

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`

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`

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`

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`

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`

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`

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_SSCALED_PACK32`

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`

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`

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`

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`

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`

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`

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_SSCALED_PACK32`

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`

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`

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`

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`

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`

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_SSCALED_PACK32`

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`

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`

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`

A one-component, 16-bit unsigned normalized format that has a single 16-bit R component.

`VK_FORMAT_R16_SNORM`

A one-component, 16-bit signed normalized format that has a single 16-bit R component.

`VK_FORMAT_R16_USCALED`

A one-component, 16-bit unsigned scaled integer format that has a single 16-bit R component.

`VK_FORMAT_R16_SSCALED`

A one-component, 16-bit signed scaled integer format that has a single 16-bit R component.

`VK_FORMAT_R16_UINT`

A one-component, 16-bit unsigned integer format that has a single 16-bit R component.

`VK_FORMAT_R16_SINT`

A one-component, 16-bit signed integer format that has a single 16-bit R component.

`VK_FORMAT_R16_SFLOAT`

A one-component, 16-bit signed floating-point format that has a single 16-bit R component.

`VK_FORMAT_R16G16_UNORM`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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`

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_SSCALED`

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`

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`

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`

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`

A one-component, 32-bit unsigned integer format that has a single 32-bit R component.

`VK_FORMAT_R32_SINT`

A one-component, 32-bit signed integer format that has a single 32-bit R component.

`VK_FORMAT_R32_SFLOAT`

A one-component, 32-bit signed floating-point format that has a single 32-bit R component.

`VK_FORMAT_R32G32_UINT`

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`

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`

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`

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`

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`

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`

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`

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`

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`

A one-component, 64-bit unsigned integer format that has a single 64-bit R component.

`VK_FORMAT_R64_SINT`

A one-component, 64-bit signed integer format that has a single 64-bit R component.

`VK_FORMAT_R64_SFLOAT`

A one-component, 64-bit signed floating-point format that has a single 64-bit R component.

`VK_FORMAT_R64G64_UINT`

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`

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`

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`

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`

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`

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`

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

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

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

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

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

A one-component, 16-bit unsigned normalized format that has a single 16-bit depth component.

VK_FORMAT_X8_D24_UNORM_PACK32

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

A one-component, 32-bit signed floating-point format that has 32-bits in the depth component.

VK_FORMAT_S8_UINT

A one-component, 8-bit unsigned integer format that has 8-bits in the stencil component.

VK_FORMAT_D16_UNORM_S8_UINT

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

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

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

A three-component, block-compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.

`VK_FORMAT_BC1_RGB_SRGB_BLOCK`

A three-component, block-compressed format where each 64-bit compressed texel block encodes a 4x4 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`

A four-component, block-compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.

`VK_FORMAT_BC1_RGBA_SRGB_BLOCK`

A four-component, block-compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

`VK_FORMAT_BC2_UNORM_BLOCK`

A four-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A four-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A four-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A four-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A one-component, block-compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized red texel data.

`VK_FORMAT_BC4_SNORM_BLOCK`

A one-component, block-compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of signed normalized red texel data.

`VK_FORMAT_BC5_UNORM_BLOCK`

A two-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A two-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A three-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 rectangle of unsigned floating-point RGB texel data.

`VK_FORMAT_BC6H_SFLOAT_BLOCK`

A three-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 rectangle of signed floating-point RGB texel data.

`VK_FORMAT_BC7_UNORM_BLOCK`

A four-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_BC7_SRGB_BLOCK`

A four-component, block-compressed format where each 128-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK`

A three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.

`VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK`

A three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4x4 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`

A four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.

`VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK`

A four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

`VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK`

A four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4x4 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`

A one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized red texel data.

`VK_FORMAT_EAC_R11_SNORM_BLOCK`

A one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4x4 rectangle of signed normalized red texel data.

`VK_FORMAT_EAC_R11G11_UNORM_BLOCK`

A two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4x4 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`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_4x4_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4x4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_5x4_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5x4 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_5x4_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5x4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_5x5_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5x5 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_5x5_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5x5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_6x5_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6x5 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_6x5_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6x5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_6x6_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6x6 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_6x6_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6x6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_8x5_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8x5 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_8x5_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8x5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_8x6_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8x6 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_8x6_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8x6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_8x8_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8x8 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_8x8_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8x8 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_10x5_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10x5 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_10x5_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10x5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_10x6_UNORM_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10x6 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_10x6_SRGB_BLOCK`

A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a

10x6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_10x8_UNORM_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10x8 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_10x8_SRGB_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10x8 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_10x10_UNORM_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10x10 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_10x10_SRGB_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10x10 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_12x10_UNORM_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12x10 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_12x10_SRGB_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12x10 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

`VK_FORMAT_ASTC_12x12_UNORM_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12x12 rectangle of unsigned normalized RGBA texel data.

`VK_FORMAT_ASTC_12x12_SRGB_BLOCK`
A four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12x12 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

#### 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 a single data element, rather than individual components occupying a single data element:

- 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_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_B10G11R11_UFLOAT_PACK32
    - VK_FORMAT_E5B9G9R9_UFLOAT_PACK32
    - VK_FORMAT_X8_D24_UNORM_PACK32

#### 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:

```
etext:VK_FORMAT_{component-format|compression-scheme}_{numeric-format}
```

The component-format specifies 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 38. Interpretation of Numeric Format*

| Numeric format | Description |
|---|---|
| `UNORM` | The components are unsigned normalized values in the range [0,1] |
| `SNORM` | The components are signed normalized values in the range [-1,1] |
| `USCALED` | The components are unsigned integer values that get converted to floating-point in the range $[0,2^n-1]$ |
| `SSCALED` | The components are signed integer values that get converted to floating-point in the range $[-2^{n-1},2^{n-1}-1]$ |
| `UINT` | The components are unsigned integer values in the range $[0,2^n-1]$ |
| `SINT` | The components are signed integer values in the range $[-2^{n-1},2^{n-1}-1]$ |
| `UFLOAT` | The components are unsigned floating-point numbers (used by packed, shared exponent, and some compressed formats) |
| `SFLOAT` | The components are signed floating-point numbers |
| `SRGB` | 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 `_BLOCK` indicates that the format is a block-compressed format, with the representation of multiple pixels encoded interdependently within a region.

*Table 39. Interpretation of Compression Scheme*

| Compression scheme | Description |
|---|---|
| `BC` | Block Compression. See Block-Compressed Image Formats. |
| `ETC2` | Ericsson Texture Compression. See ETC Compressed Image Formats. |
| `EAC` | ETC2 Alpha Compression. See ETC Compressed Image Formats. |
| `ASTC` | Adaptive Scalable Texture Compression (LDR Profile). See ASTC Compressed Image Formats. |

#### Representation

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 an *element size*, the number of bytes used to stored one element or one compressed block, with the value of the element size listed in VkFormat.

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 40. 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_*` |
| R | | | | | | | | | | | | | | | | `VK_FORMAT_R16_*` |
| R | | G | | | | | | | | | | | | | | `VK_FORMAT_R16G16_*` |
| R | | G | | B | | | | | | | | | | | | `VK_FORMAT_R16G16B16_*` |
| R | | G | | B | | A | | | | | | | | | | `VK_FORMAT_R16G16B16A16_*` |
| R | | | | | | | | | | | | | | | | `VK_FORMAT_R32_*` |
| 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_*` |
| `VK_FORMAT_R64G64B64_*` as `VK_FORMAT_R64G64_*` but with B in bytes 16-23 | | | | | | | | | | | | | | | | |
| `VK_FORMAT_R64G64B64A64_*` as `VK_FORMAT_R64G64B64_*` 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 41. Bit mappings for packed 8-bit formats*

| Bit | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| `VK_FORMAT_R4G4_UNORM_PACK8` | | | | | | | |
| R | | | | G | | | |
| 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 |

*Table 42. Bit mappings for packed 16-bit formats*

| Bit | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| `VK_FORMAT_R4G4B4A4_UNORM_PACK16` | | | | | | | | | | | | | | | |
| R | | | | G | | | | B | | | | A | | | |
| 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 |
| `VK_FORMAT_B4G4R4A4_UNORM_PACK16` | | | | | | | | | | | | | | | |

| Bit | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | | | | G | | | | R | | | | A | | | |
| 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 |
| VK_FORMAT_R5G6B5_UNORM_PACK16 | | | | | | | | | | | | | | | |
| R | | | | | G | | | | | | B | | | | |
| 4 | 3 | 2 | 1 | 0 | 5 | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 |
| VK_FORMAT_B5G6R5_UNORM_PACK16 | | | | | | | | | | | | | | | |
| B | | | | | G | | | | | | R | | | | |
| 4 | 3 | 2 | 1 | 0 | 5 | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 |
| VK_FORMAT_R5G5B5A1_UNORM_PACK16 | | | | | | | | | | | | | | | |
| R | | | | | G | | | | | B | | | | | A |
| 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 | 0 |
| VK_FORMAT_B5G5R5A1_UNORM_PACK16 | | | | | | | | | | | | | | | |
| B | | | | | G | | | | | R | | | | | A |
| 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 | 0 |
| VK_FORMAT_A1R5G5B5_UNORM_PACK16 | | | | | | | | | | | | | | | |
| A | R | | | | | G | | | | | B | | | | |
| 0 | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 |

*Table 43. 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 | | | | | | | | | |
| 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 | | | | | | | | | |
| 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 | | | | | | | | | |
| 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 |

### 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).

### Format Compatibility Classes

Uncompressed color formats are *compatible* with each other if they occupy the same number of bits per data element. 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 44. Compatible formats*

| Class | Formats |
|---|---|
| 8-bit | VK_FORMAT_R4G4_UNORM_PACK8,<br>VK_FORMAT_R8_UNORM,<br>VK_FORMAT_R8_SNORM,<br>VK_FORMAT_R8_USCALED,<br>VK_FORMAT_R8_SSCALED,<br>VK_FORMAT_R8_UINT,<br>VK_FORMAT_R8_SINT,<br>VK_FORMAT_R8_SRGB |
| 16-bit | VK_FORMAT_R4G4B4A4_UNORM_PACK16,<br>VK_FORMAT_B4G4R4A4_UNORM_PACK16,<br>VK_FORMAT_R5G6B5_UNORM_PACK16,<br>VK_FORMAT_B5G6R5_UNORM_PACK16,<br>VK_FORMAT_R5G5B5A1_UNORM_PACK16,<br>VK_FORMAT_B5G5R5A1_UNORM_PACK16,<br>VK_FORMAT_A1R5G5B5_UNORM_PACK16,<br>VK_FORMAT_R8G8_UNORM,<br>VK_FORMAT_R8G8_SNORM,<br>VK_FORMAT_R8G8_USCALED,<br>VK_FORMAT_R8G8_SSCALED,<br>VK_FORMAT_R8G8_UINT,<br>VK_FORMAT_R8G8_SINT,<br>VK_FORMAT_R8G8_SRGB,<br>VK_FORMAT_R16_UNORM,<br>VK_FORMAT_R16_SNORM,<br>VK_FORMAT_R16_USCALED,<br>VK_FORMAT_R16_SSCALED,<br>VK_FORMAT_R16_UINT,<br>VK_FORMAT_R16_SINT,<br>VK_FORMAT_R16_SFLOAT |

| Class | Formats |
| --- | --- |
| 24-bit | VK_FORMAT_R8G8B8_UNORM,<br>VK_FORMAT_R8G8B8_SNORM,<br>VK_FORMAT_R8G8B8_USCALED,<br>VK_FORMAT_R8G8B8_SSCALED,<br>VK_FORMAT_R8G8B8_UINT,<br>VK_FORMAT_R8G8B8_SINT,<br>VK_FORMAT_R8G8B8_SRGB,<br>VK_FORMAT_B8G8R8_UNORM,<br>VK_FORMAT_B8G8R8_SNORM,<br>VK_FORMAT_B8G8R8_USCALED,<br>VK_FORMAT_B8G8R8_SSCALED,<br>VK_FORMAT_B8G8R8_UINT,<br>VK_FORMAT_B8G8R8_SINT,<br>VK_FORMAT_B8G8R8_SRGB |

| Class | Formats |
| --- | --- |
| 32-bit | VK_FORMAT_R8G8B8A8_UNORM,<br>VK_FORMAT_R8G8B8A8_SNORM,<br>VK_FORMAT_R8G8B8A8_USCALED,<br>VK_FORMAT_R8G8B8A8_SSCALED,<br>VK_FORMAT_R8G8B8A8_UINT,<br>VK_FORMAT_R8G8B8A8_SINT,<br>VK_FORMAT_R8G8B8A8_SRGB,<br>VK_FORMAT_B8G8R8A8_UNORM,<br>VK_FORMAT_B8G8R8A8_SNORM,<br>VK_FORMAT_B8G8R8A8_USCALED,<br>VK_FORMAT_B8G8R8A8_SSCALED,<br>VK_FORMAT_B8G8R8A8_UINT,<br>VK_FORMAT_B8G8R8A8_SINT,<br>VK_FORMAT_B8G8R8A8_SRGB,<br>VK_FORMAT_A8B8G8R8_UNORM_PACK32,<br>VK_FORMAT_A8B8G8R8_SNORM_PACK32,<br>VK_FORMAT_A8B8G8R8_USCALED_PACK32,<br>VK_FORMAT_A8B8G8R8_SSCALED_PACK32,<br>VK_FORMAT_A8B8G8R8_UINT_PACK32,<br>VK_FORMAT_A8B8G8R8_SINT_PACK32,<br>VK_FORMAT_A8B8G8R8_SRGB_PACK32,<br>VK_FORMAT_A2R10G10B10_UNORM_PACK32,<br>VK_FORMAT_A2R10G10B10_SNORM_PACK32,<br>VK_FORMAT_A2R10G10B10_USCALED_PACK32,<br>VK_FORMAT_A2R10G10B10_SSCALED_PACK32,<br>VK_FORMAT_A2R10G10B10_UINT_PACK32,<br>VK_FORMAT_A2R10G10B10_SINT_PACK32,<br>VK_FORMAT_A2B10G10R10_UNORM_PACK32,<br>VK_FORMAT_A2B10G10R10_SNORM_PACK32,<br>VK_FORMAT_A2B10G10R10_USCALED_PACK32,<br>VK_FORMAT_A2B10G10R10_SSCALED_PACK32,<br>VK_FORMAT_A2B10G10R10_UINT_PACK32,<br>VK_FORMAT_A2B10G10R10_SINT_PACK32,<br>VK_FORMAT_R16G16_UNORM,<br>VK_FORMAT_R16G16_SNORM,<br>VK_FORMAT_R16G16_USCALED,<br>VK_FORMAT_R16G16_SSCALED,<br>VK_FORMAT_R16G16_UINT,<br>VK_FORMAT_R16G16_SINT,<br>VK_FORMAT_R16G16_SFLOAT,<br>VK_FORMAT_R32_UINT,<br>VK_FORMAT_R32_SINT,<br>VK_FORMAT_R32_SFLOAT,<br>VK_FORMAT_B10G11R11_UFLOAT_PACK32,<br>VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 |

| Class | Formats |
| --- | --- |
| 48-bit | VK_FORMAT_R16G16B16_UNORM,<br>VK_FORMAT_R16G16B16_SNORM,<br>VK_FORMAT_R16G16B16_USCALED,<br>VK_FORMAT_R16G16B16_SSCALED,<br>VK_FORMAT_R16G16B16_UINT,<br>VK_FORMAT_R16G16B16_SINT,<br>VK_FORMAT_R16G16B16_SFLOAT |
| 64-bit | VK_FORMAT_R16G16B16A16_UNORM,<br>VK_FORMAT_R16G16B16A16_SNORM,<br>VK_FORMAT_R16G16B16A16_USCALED,<br>VK_FORMAT_R16G16B16A16_SSCALED,<br>VK_FORMAT_R16G16B16A16_UINT,<br>VK_FORMAT_R16G16B16A16_SINT,<br>VK_FORMAT_R16G16B16A16_SFLOAT,<br>VK_FORMAT_R32G32_UINT,<br>VK_FORMAT_R32G32_SINT,<br>VK_FORMAT_R32G32_SFLOAT,<br>VK_FORMAT_R64_UINT,<br>VK_FORMAT_R64_SINT,<br>VK_FORMAT_R64_SFLOAT |
| 96-bit | VK_FORMAT_R32G32B32_UINT,<br>VK_FORMAT_R32G32B32_SINT,<br>VK_FORMAT_R32G32B32_SFLOAT |
| 128-bit | VK_FORMAT_R32G32B32A32_UINT,<br>VK_FORMAT_R32G32B32A32_SINT,<br>VK_FORMAT_R32G32B32A32_SFLOAT,<br>VK_FORMAT_R64G64_UINT,<br>VK_FORMAT_R64G64_SINT,<br>VK_FORMAT_R64G64_SFLOAT |
| 192-bit | VK_FORMAT_R64G64B64_UINT,<br>VK_FORMAT_R64G64B64_SINT,<br>VK_FORMAT_R64G64B64_SFLOAT |
| 256-bit | VK_FORMAT_R64G64B64A64_UINT,<br>VK_FORMAT_R64G64B64A64_SINT,<br>VK_FORMAT_R64G64B64A64_SFLOAT |
| BC1_RGB | VK_FORMAT_BC1_RGB_UNORM_BLOCK,<br>VK_FORMAT_BC1_RGB_SRGB_BLOCK |
| BC1_RGBA | VK_FORMAT_BC1_RGBA_UNORM_BLOCK,<br>VK_FORMAT_BC1_RGBA_SRGB_BLOCK |
| BC2 | VK_FORMAT_BC2_UNORM_BLOCK,<br>VK_FORMAT_BC2_SRGB_BLOCK |
| BC3 | VK_FORMAT_BC3_UNORM_BLOCK,<br>VK_FORMAT_BC3_SRGB_BLOCK |
| BC4 | VK_FORMAT_BC4_UNORM_BLOCK,<br>VK_FORMAT_BC4_SNORM_BLOCK |

| Class | Formats |
| --- | --- |
| BC5 | VK_FORMAT_BC5_UNORM_BLOCK, VK_FORMAT_BC5_SNORM_BLOCK |
| BC6H | VK_FORMAT_BC6H_UFLOAT_BLOCK, VK_FORMAT_BC6H_SFLOAT_BLOCK |
| BC7 | VK_FORMAT_BC7_UNORM_BLOCK, VK_FORMAT_BC7_SRGB_BLOCK |
| ETC2_RGB | VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK |
| ETC2_RGBA | VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK |
| ETC2_EAC_RGBA | VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK |
| EAC_R | VK_FORMAT_EAC_R11_UNORM_BLOCK, VK_FORMAT_EAC_R11_SNORM_BLOCK |
| EAC_RG | VK_FORMAT_EAC_R11G11_UNORM_BLOCK, VK_FORMAT_EAC_R11G11_SNORM_BLOCK |
| ASTC_4x4 | VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SRGB_BLOCK |
| ASTC_5x4 | VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SRGB_BLOCK |
| ASTC_5x5 | VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SRGB_BLOCK |
| ASTC_6x5 | VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SRGB_BLOCK |
| ASTC_6x6 | VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SRGB_BLOCK |
| ASTC_8x5 | VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SRGB_BLOCK |
| ASTC_8x6 | VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SRGB_BLOCK |
| ASTC_8x8 | VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SRGB_BLOCK |
| ASTC_10x5 | VK_FORMAT_ASTC_10x5_UNORM_BLOCK, VK_FORMAT_ASTC_10x5_SRGB_BLOCK |
| ASTC_10x6 | VK_FORMAT_ASTC_10x6_UNORM_BLOCK, VK_FORMAT_ASTC_10x6_SRGB_BLOCK |
| ASTC_10x8 | VK_FORMAT_ASTC_10x8_UNORM_BLOCK, VK_FORMAT_ASTC_10x8_SRGB_BLOCK |
| ASTC_10x10 | VK_FORMAT_ASTC_10x10_UNORM_BLOCK, VK_FORMAT_ASTC_10x10_SRGB_BLOCK |
| ASTC_12x10 | VK_FORMAT_ASTC_12x10_UNORM_BLOCK, VK_FORMAT_ASTC_12x10_SRGB_BLOCK |

| Class | Formats |
|---|---|
| ASTC_12x12 | VK_FORMAT_ASTC_12x12_UNORM_BLOCK, VK_FORMAT_ASTC_12x12_SRGB_BLOCK |
| D16 | VK_FORMAT_D16_UNORM |
| D24 | VK_FORMAT_X8_D24_UNORM_PACK32 |
| D32 | VK_FORMAT_D32_SFLOAT |
| S8 | VK_FORMAT_S8_UINT |
| D16S8 | VK_FORMAT_D16_UNORM_S8_UINT |
| D24S8 | VK_FORMAT_D24_UNORM_S8_UINT |
| D32S8 | VK_FORMAT_D32_SFLOAT_S8_UINT |

### 32.3.2. 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 pointer to a `VkFormatProperties` structure

The `VkFormatProperties` structure is defined as:

```
typedef struct VkFormatProperties {
    VkFormatFeatureFlags    linearTilingFeatures;
    VkFormatFeatureFlags    optimalTilingFeatures;
    VkFormatFeatureFlags    bufferFeatures;
} VkFormatProperties;
```

- `linearTilingFeatures` describes the features supported by `VK_IMAGE_TILING_LINEAR`.
- `optimalTilingFeatures` describes the features supported by `VK_IMAGE_TILING_OPTIMAL`.

- `bufferFeatures` describes the features supported by buffers.

Supported features are described as a set of VkFormatFeatureFlagBits:

```
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_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG = 0x00002000,
    VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR = 0x00004000,
    VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR = 0x00008000,
} VkFormatFeatureFlagBits;
```

The `linearTilingFeatures` and `optimalTilingFeatures` members of the `VkFormatProperties` structure describe what features are supported by `VK_IMAGE_TILING_LINEAR` and `VK_IMAGE_TILING_OPTIMAL` images, respectively.

The following bits **may** be set in `linearTilingFeatures` and `optimalTilingFeatures`, indicating they are supported by images or image views created with the queried vkGetPhysicalDeviceFormatProperties ::`format`:

`VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`
: `VkImageView` **can** be sampled from. See sampled images section.

`VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`
: `VkImageView` **can** be used as storage image. See storage images section.

`VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
: `VkImageView` **can** be used as storage image that supports atomic operations.

`VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`
: `VkImageView` **can** be used as a framebuffer color attachment and as an input attachment.

`VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`
: `VkImageView` **can** be used as a framebuffer color attachment that supports blending and as an input attachment.

`VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`
: `VkImageView` **can** be used as a framebuffer depth/stencil attachment and as an input attachment.

VK_FORMAT_FEATURE_BLIT_SRC_BIT

VkImage **can** be used as srcImage for the vkCmdBlitImage command.

VK_FORMAT_FEATURE_BLIT_DST_BIT

VkImage **can** be used as dstImage for the vkCmdBlitImage command.

VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

If VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT is also set, VkImageView **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, VkImage 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 indicates 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_KHR

VkImage **can** be used as a source image for copy commands.

VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR

VkImage **can** be used as a destination image for copy commands and clear commands.

VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG

VkImage **can** be used with a sampler that has either of magFilter or minFilter set to VK_FILTER_CUBIC_IMG, or be the source image for a blit with filter set to VK_FILTER_CUBIC_IMG. 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 indicates that the depth aspect is cubic filterable.

The following features **may** appear in bufferFeatures, indicating they are supported by buffers or buffer views created with the queried vkGetPhysicalDeviceFormatProperties::format:

VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT

Format **can** be used to create a VkBufferView that **can** be bound to a VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER descriptor.

VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT

Format **can** be used to create a VkBufferView that **can** be bound to a VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER descriptor.

VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT

Atomic operations are supported on VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER with this format.

VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT

Format **can** be used as a vertex attribute format (VkVertexInputAttributeDescription::format).

> *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-compression format, then buffers **must** not support any features for the format.

To query supported format features which are properties of the physical device, call:

```
void vkGetPhysicalDeviceFormatProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkFormatProperties2KHR*                     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 VkFormatProperties2KHR structure in which physical device properties for format are returned.

vkGetPhysicalDeviceFormatProperties2KHR 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 pointer to a VkFormatProperties2KHR structure

The VkFormatProperties2KHR structure is defined as:

```
typedef struct VkFormatProperties2KHR {
    VkStructureType       sType;
    void*                 pNext;
    VkFormatProperties    formatProperties;
} VkFormatProperties2KHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to an extension-specific structure.
- formatProperties is a structure of type VkFormatProperties describing features supported by the requested format.

### 32.3.3. 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.

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_KHR` and `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`.

*Table 45. Key for format feature tables*

| | |
|---|---|
| ✓ | This feature **must** be supported on the named format |
| † | This feature **must** be supported on at least some of the named formats, with more information in the table where the symbol appears |

*Table 46. Feature bits in `optimalTilingFeatures`*

| |
|---|
| `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR` |
| `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR` |
| `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` |
| `VK_FORMAT_FEATURE_BLIT_SRC_BIT` |
| `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` |
| `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT` |
| `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT` |
| `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` |
| `VK_FORMAT_FEATURE_BLIT_DST_BIT` |
| `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT` |
| `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT` |

*Table 47. 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 48. Mandatory format support: sub-byte 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_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 49. 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 | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| **Format** | ↓ | | | | | | | | | | | | |
| VK_FORMAT_R8_UNORM | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | | ✓ | ✓ | | |
| VK_FORMAT_R8_SNORM | ✓ | ✓ | ✓ | | | | | | | ✓ | ✓ | | |
| VK_FORMAT_R8_USCALED | | | | | | | | | | | | | |
| VK_FORMAT_R8_SSCALED | | | | | | | | | | | | | |
| 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_SSCALED | | | | | | | | | | | | | |
| 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_SSCALED | | | | | | | | | | | | | |
| 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_SSCALED | | | | | | | | | | | | | |
| VK_FORMAT_B8G8R8_UINT | | | | | | | | | | | | | |
| VK_FORMAT_B8G8R8_SINT | | | | | | | | | | | | | |
| VK_FORMAT_B8G8R8_SRGB | | | | | | | | | | | | | |

*Table 50. Mandatory format support: 4 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 | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| **Format** | ↓ | | | | | | | | | | | | |
| 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 51. Mandatory format support: 10-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_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 | | | | | | | | | | | | | |

*Table 52. 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_SSCALED | | | | | | | | | | | | | |
| 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_SSCALED | | | | | | | | | | | | | |
| 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_SSCALED | | | | | | | | | | | | | |
| 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_SSCALED | | | | | | | | | | | | | |
| VK_FORMAT_R16G16B16A16_UINT | ✓ | ✓ | | ✓ | | ✓ | ✓ | | | ✓ | ✓ | ✓ | |
| VK_FORMAT_R16G16B16A16_SINT | ✓ | ✓ | | ✓ | | ✓ | ✓ | | | ✓ | ✓ | ✓ | |
| VK_FORMAT_R16G16B16A16_SFLOAT | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | |

*Table 53. Mandatory format support: 32-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_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 54. Mandatory format support: 64-bit/uneven 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_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 55. Mandatory format support: depth/stencil with VkImageType* VK_IMAGE_TYPE_2D

| 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_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 **must** be supported for at least one of VK_FORMAT_X8_D24_UNORM_PACK32 and VK_FORMAT_D32_SFLOAT, and **must** be supported for at least one of VK_FORMAT_D24_UNORM_S8_UINT and VK_FORMAT_D32_SFLOAT_S8_UINT.

*Table 56. Mandatory format support: BC compressed formats with VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D*

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | 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_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 | † | † | † | | | | | | | | | | |

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 57. Mandatory format support: ETC2 and EAC compressed formats with VkImageType VK_IMAGE_TYPE_2D*

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | 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_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 | † | † | † | | | | | | | | | | |

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 58. Mandatory format support: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D*

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | 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_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 | † | † | † | | | | | | | | | | |
| 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. | | | | | | | | | | | | | |

VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG **must** be supported 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_SSCALED_PACK32
- VK_FORMAT_A8B8G8R8_UINT_PACK32
- VK_FORMAT_A8B8G8R8_SINT_PACK32
- VK_FORMAT_A8B8G8R8_SRGB_PACK32

If ETC2 compressed formats are supported, the following additional formats **must** support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG`:

- 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

## 32.4. Additional Image Capabilities

In addition to the minimum capabilities described in the previous sections (Limits and Formats), implementations **may** support additional capabilities for certain types of images. For example, larger dimensions or additional sample counts for certain image types, or additional capabilities for *linear* tiling format images.

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 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`.
- `pImageFormatProperties` points to an instance of the 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 (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 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** either be equal to 1 (valid only if `tiling` is `VK_IMAGE_TILING_LINEAR`) or be equal to $\log_2(\max(\texttt{width}, \texttt{height}, \texttt{depth})) + 1$. `width`, `height`, and `depth` are taken from the corresponding members of `maxExtent`.
- `maxArrayLayers` is the maximum number of array layers. `maxArrayLayers` **must** either be equal to 1 or be greater than or equal to the `maxImageArrayLayers` member of VkPhysicalDeviceLimits. A value of 1 is valid only if `tiling` is `VK_IMAGE_TILING_LINEAR` or if `type` is `VK_IMAGE_TYPE_3D`.
- `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 or the image will be invalid. 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.

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,
    VkExternalImageFormatPropertiesNV*          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` points to an instance of the 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 `pExternalImageFormatProperties` 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 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

The bits in VkExternalMemoryFeatureFlagBitsNV are defined as follows:

```
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,
} VkExternalMemoryFeatureFlagBitsNV;
```

- VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV: 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: The implementation supports exporting handles of the specified type.
- VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_NV: The implementation supports importing handles of the specified type.

To query additional capabilities specific to image types, call:

```
VkResult vkGetPhysicalDeviceImageFormatProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    const VkPhysicalDeviceImageFormatInfo2KHR*  pImageFormatInfo,
    VkImageFormatProperties2KHR*                pImageFormatProperties);
```

- `physicalDevice` is the physical device from which to query the image capabilities.
- `pImageFormatInfo` points to an instance of the VkPhysicalDeviceImageFormatInfo2KHR structure, describing the parameters that would be consumed by vkCreateImage.
- `pImageFormatProperties` points to an instance of the VkImageFormatProperties2KHR structure in which capabilities are returned.

`vkGetPhysicalDeviceImageFormatProperties2KHR` behaves similarly to vkGetPhysicalDeviceImageFormatProperties, with the ability to return extended information in a `pNext` chain of output structures.

If the loader implementation emulates `vkGetPhysicalDeviceImageFormatProperties2KHR` on a device that does not support the extension, and the query involves a structure the loader does not support, `vkGetPhysicalDeviceImageFormatProperties2KHR` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pImageFormatInfo` **must** be a pointer to a valid `VkPhysicalDeviceImageFormatInfo2KHR` structure
- `pImageFormatProperties` **must** be a pointer to a `VkImageFormatProperties2KHR` 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 `VkPhysicalDeviceImageFormatInfo2KHR` structure is defined as:

```
typedef struct VkPhysicalDeviceImageFormatInfo2KHR {
    VkStructureType       sType;
    const void*           pNext;
    VkFormat              format;
    VkImageType           type;
    VkImageTiling         tiling;
    VkImageUsageFlags     usage;
    VkImageCreateFlags    flags;
} VkPhysicalDeviceImageFormatInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure. The `pNext` chain of `VkPhysicalDeviceImageFormatInfo2KHR` is used to provide additional image parameters to `vkGetPhysicalDeviceImageFormatProperties2KHR`.
- `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`.

The members of `VkPhysicalDeviceImageFormatInfo2KHR` correspond to the arguments to vkGetPhysicalDeviceImageFormatProperties, with `sType` and `pNext` added for extensibility.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2_KHR`
- `pNext` **must** be `NULL` or a pointer to a valid instance of VkPhysicalDeviceExternalImageFormatInfoKHX
- `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 `VkImageFormatProperties2KHR` structure is defined as:

```
typedef struct VkImageFormatProperties2KHR {
    VkStructureType            sType;
    void*                      pNext;
    VkImageFormatProperties    imageFormatProperties;
} VkImageFormatProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure. The `pNext` chain of `VkImageFormatProperties2KHR` is used to allow the specification of additional capabilities to be returned from `vkGetPhysicalDeviceImageFormatProperties2KHR`.
- `imageFormatProperties` is an instance of a VkImageFormatProperties structure in which capabilities are returned.

If the combination of parameters to `vkGetPhysicalDeviceImageFormatProperties2KHR` is not supported by the implementation for use in vkCreateImage, then all members of `imageFormatProperties` will be filled with zero.

To determine if texture gather functions that take explicit LOD and/or bias argument values can be used with a given image format, add VkImageFormatProperties2KHR to the `pNext` chain of the VkPhysicalDeviceImageFormatInfo2KHR structure and VkTextureLODGatherFormatPropertiesAMD to the `pNext` chain of the VkImageFormatProperties2KHR structure.

The `VkTextureLODGatherFormatPropertiesAMD` 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_lod` extension. (see VK_AMD_texture_gather_bias_lod). This field is set by the implementation. User-specified value is ignored.

To determine the image capabilities compatible with an external memory handle type, add VkPhysicalDeviceExternalImageFormatInfoKHX to the `pNext` chain of the VkPhysicalDeviceImageFormatInfo2KHR structure and `VkExternalImageFormatPropertiesKHX` to the `pNext` chain of the VkImageFormatProperties2KHR structure.

The `VkPhysicalDeviceExternalImageFormatInfoKHX` structure is defined as:

```
typedef struct VkPhysicalDeviceExternalImageFormatInfoKHX {
    VkStructureType                           sType;
    const void*                               pNext;
    VkExternalMemoryHandleTypeFlagBitsKHX     handleType;
} VkPhysicalDeviceExternalImageFormatInfoKHX;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to an extension-specific structure.
- handleType is a bit indicating a memory handle type that will be used with the memory associated with the image. Bits which can be set include:

```
typedef enum VkExternalMemoryHandleTypeFlagBitsKHX {
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHX = 0x00000001,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX = 0x00000002,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHX = 0x00000004,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHX = 0x00000008,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT_KHX = 0x00000010,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT_KHX = 0x00000020,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT_KHX = 0x00000040,
} VkExternalMemoryHandleTypeFlagBitsKHX;
```

  - VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHX is 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_WIN32_BIT_KHX is 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_KHX is 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 own a reference to the underlying memory resource represented 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_KHX is 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_KHX is a global share handle returned by IDXGIResource::GetSharedHandle referring to a Direct3D 10 or 11 texture resource. It does not own 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_KHX is 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_KHX is 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.

If handleType is 0, vkGetPhysicalDeviceImageFormatProperties2KHR will behave as if VkPhysicalDeviceExternalImageFormatInfoKHX was not present and VkExternalImageFormatPropertiesKHX will be ignored.

If handleType is not compatible with the format, type, tiling, usage, and flags specified in VkPhysicalDeviceImageFormatInfo2KHR, then vkGetPhysicalDeviceImageFormatProperties2KHR returns VK_ERROR_FORMAT_NOT_SUPPORTED.

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO_KHX
- If handleType is not 0, handleType **must** be a valid VkExternalMemoryHandleTypeFlagBitsKHX value

The VkExternalImageFormatPropertiesKHX structure is defined as:

```
typedef struct VkExternalImageFormatPropertiesKHX {
    VkStructureType                 sType;
    void*                           pNext;
    VkExternalMemoryPropertiesKHX   externalMemoryProperties;
} VkExternalImageFormatPropertiesKHX;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to an extension-specific structure.
- externalMemoryProperties is an instance of the VkExternalMemoryPropertiesKHX structure specifying various capabilities of the external handle type when used with the specified image creation parameters.

The VkExternalMemoryPropertiesKHX structure is defined as:

```
typedef struct VkExternalMemoryPropertiesKHX {
    VkExternalMemoryFeatureFlagsKHX       externalMemoryFeatures;
    VkExternalMemoryHandleTypeFlagsKHX    exportFromImportedHandleTypes;
    VkExternalMemoryHandleTypeFlagsKHX    compatibleHandleTypes;
} VkExternalMemoryPropertiesKHX;
```

- externalMemoryFeatures is a bitmask describing the features of handleType. See VkExternalMemoryFeatureFlagBitsKHX below for a description of the possible bits.

- exportFromImportedHandleTypes is a bitmask specifying handle types that can be used to import objects from which handleType can be exported.
- compatibleHandleTypes is a bitmask 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 VkImageFormatProperties2KHR will be the same when VkPhysicalDeviceExternalImageFormatInfoKHX::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.

The features of an external memory handle type are returned in VkExternalMemoryPropertiesKHX ::externalMemoryFeatures. Bits which **may** be set include:

```
typedef enum VkExternalMemoryFeatureFlagBitsKHX {
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_KHX = 0x00000001,
    VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_KHX = 0x00000002,
    VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_KHX = 0x00000004,
} VkExternalMemoryFeatureFlagBitsKHX;
```

These bits have the following meanings:

- VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_KHX indicates that images or buffers created with the specified parameters and handle type **must** use the mechanisms defined in the VK_NV_dedicated_allocation to create (or import) a dedicated allocation for the image or buffer.
- VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_KHX indicates handles of this type **can** be exported from Vulkan memory objects.
- VK_INTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_KHX indicates 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_KHX for the following external handle types:

- VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHX
- VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT_KHX
- VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT_KHX

### 32.4.1. 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

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, 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.

### 32.4.2. 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 ≥ 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

## 32.5. Additional Buffer Capabilities

In addition to the capabilities described in the previous sections (Limits and Formats), implementations may support additional buffer capabilities.

To query the external handle types supported by buffers, call:

```
void vkGetPhysicalDeviceExternalBufferPropertiesKHX(
    VkPhysicalDevice                            physicalDevice,
    const VkPhysicalDeviceExternalBufferInfoKHX* pExternalBufferInfo,
    VkExternalBufferPropertiesKHX*              pExternalBufferProperties);
```

- physicalDevice is the physical device from which to query the buffer capabilities.
- pExternalBufferInfo points to an instance of the VkPhysicalDeviceExternalBufferInfoKHX structure, describing the parameters that would be consumed by vkCreateBuffer.
- pExternalBufferProperties points to an instance of the VkExternalBufferPropertiesKHX structure in which capabilities are returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pExternalBufferInfo` **must** be a pointer to a valid `VkPhysicalDeviceExternalBufferInfoKHX` structure
- `pExternalBufferProperties` **must** be a pointer to a `VkExternalBufferPropertiesKHX` structure

The `VkPhysicalDeviceExternalBufferInfoKHX` structure is defined as:

```
typedef struct VkPhysicalDeviceExternalBufferInfoKHX {
    VkStructureType                           sType;
    const void*                               pNext;
    VkBufferCreateFlags                       flags;
    VkBufferUsageFlags                        usage;
    VkExternalMemoryHandleTypeFlagBitsKHX     handleType;
} VkPhysicalDeviceExternalBufferInfoKHX;
```

- `sType` is the type of this structure
- `pNext` is NULL or a pointer to an extension-specific structure.
- `flags` is a bitmask describing additional parameters of the buffer, corresponding to VkBufferCreateInfo::`flags`.
- `usage` is a bitmask describing the intended usage of the buffer, corresponding to VkBufferCreateInfo::`usage`.
- `handleType` is a bit indicating a memory handle type that will be used with the memory associated with the buffer. See VkExternalMemoryHandleTypeFlagBitsKHX for details.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO_KHX`
- `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 VkExternalMemoryHandleTypeFlagBitsKHX value

The `VkExternalBufferPropertiesKHX` structure is defined as:

```
typedef struct VkExternalBufferPropertiesKHX {
    VkStructureType                 sType;
    void*                           pNext;
    VkExternalMemoryPropertiesKHX   externalMemoryProperties;
} VkExternalBufferPropertiesKHX;
```

- `sType` is the type of this structure
- `pNext` is NULL or a pointer to an extension-specific structure.
- `externalMemoryProperties` is an instance of the VkExternalMemoryPropertiesKHX structure specifying various capabilities of the external handle type when used with the specified buffer creation parameters.

## 32.6. Optional Semaphore Capabilities

Semaphores **may** support import and export of external semaphore handles. To query the external handle types supported by semaphores, call:

```
void vkGetPhysicalDeviceExternalSemaphorePropertiesKHX(
    VkPhysicalDevice                            physicalDevice,
    const VkPhysicalDeviceExternalSemaphoreInfoKHX* pExternalSemaphoreInfo,
    VkExternalSemaphorePropertiesKHX*           pExternalSemaphoreProperties);
```

- `physicalDevice` is the physical device from which to query the semaphore capabilities.
- `pExternalSemaphoreInfo` points to an instance of the VkPhysicalDeviceExternalSemaphoreInfoKHX structure, describing the parameters that would be consumed by vkCreateSemaphore.
- `pExternalSemaphoreProperties` points to an instance of the VkExternalSemaphorePropertiesKHX structure in which capabilities are returned.

### Valid Usage (Implicit)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `pExternalSemaphoreInfo` **must** be a pointer to a valid `VkPhysicalDeviceExternalSemaphoreInfoKHX` structure
- `pExternalSemaphoreProperties` **must** be a pointer to a `VkExternalSemaphorePropertiesKHX` structure

The `VkPhysicalDeviceExternalSemaphoreInfoKHX` structure is defined as:

```
typedef struct VkPhysicalDeviceExternalSemaphoreInfoKHX {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkExternalSemaphoreHandleTypeFlagBitsKHX        handleType;
} VkPhysicalDeviceExternalSemaphoreInfoKHX;
```

- `sType` is the type of this structure
- `pNext` is NULL or a pointer to an extension-specific structure.
- `handleType` is a bit indicating an external semaphore handle type for which capabilities will be returned. Bits which can be set include:

```
typedef enum VkExternalSemaphoreHandleTypeFlagBitsKHX {
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT_KHX = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX = 0x00000002,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHX = 0x00000004,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT_KHX = 0x00000008,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_FENCE_FD_BIT_KHX = 0x00000010,
} VkExternalSemaphoreHandleTypeFlagBitsKHX;
```

  - `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT_KHX` is 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_WIN32_BIT_KHX` is 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_KHX` is 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 own a reference to the underlying synchronization primitive represented 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_KHX` is 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_FENCE_FD_BIT_KHX` is a POSIX file descriptor handle to a Linux or Android Fence object. It can be used with any native API accepting a valid fence object file descriptor 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 fence FD supported by the native system they are running on.

*Note*

Applications **must** not perform operations on handles of type VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_FENCE_FD_BIT_KHX that are specific to only one of Linux Fences or Android Fences. Implementations are free to return either type of file descriptor using this handle type. While most userspace functions accepting a fence FD are compatible with either type of fence, attempting to use the file descriptors directly with system calls compatible with only one of the fence types **may** result in undefined behavior.

### Valid Usage (Implicit)

- sType **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO_KHX
- pNext **must** be NULL
- handleType **must** be a valid VkExternalSemaphoreHandleTypeFlagBitsKHX value

The VkExternalSemaphorePropertiesKHX structure is defined as:

```
typedef struct VkExternalSemaphorePropertiesKHX {
    VkStructureType                           sType;
    void*                                     pNext;
    VkExternalSemaphoreHandleTypeFlagsKHX     exportFromImportedHandleTypes;
    VkExternalSemaphoreHandleTypeFlagsKHX     compatibleHandleTypes;
    VkExternalSemaphoreFeatureFlagsKHX        externalSemaphoreFeatures;
} VkExternalSemaphorePropertiesKHX;
```

- exportFromImportedHandleTypes is a bitmask specifying handle types that can be used to import objects from which handleType can be exported.
- compatibleHandleTypes is a bitmask specifying handle types which can be specified at the same time as handleType when creating a semaphore.
- externalSemaphoreFeatures is a bitmask describing the features of handleType. See VkExternalSemaphoreFeatureFlagBitsKHX below for a description of the possible bits.

If handleType is not supported by the implementation, then VkExternalSemaphorePropertiesKHX ::externalSemaphoreFeatures will be set to zero.

Features which a semaphore external handle type may support include:

```
typedef enum VkExternalSemaphoreFeatureFlagBitsKHX {
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT_KHX = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT_KHX = 0x00000002,
} VkExternalSemaphoreFeatureFlagBitsKHX;
```

- VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT_KHX indicates handles of this type **can** be exported from Vulkan semaphore objects.

- `VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT_KHX` indicates handles of this type **can** be imported as Vulkan semaphore objects.

# Chapter 33. 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_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_DESCRIPTOR_UPDATE_TEMPLATE_KHR = 1000085000,
    VK_OBJECT_TYPE_OBJECT_TABLE_NVX = 1000086000,
    VK_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NVX = 1000086001,
} VkObjectType;
```

If this Specification was generated with any such extensions included, they will be described in the remainder of this chapter.

# 33.1. 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.

## 33.1.1. 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,
    VkDebugMarkerObjectNameInfoEXT*             pNameInfo);
```

- `device` is the device that created the object.
- `pNameInfo` is a pointer to an instance of the VkDebugMarkerObjectNameInfoEXT structure specifying the parameters of the name to set on the object.

### Valid Usage

- `pNameInfo.object` **must** be a Vulkan object

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pNameInfo` **must** be a pointer to a `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, pName **should** be set to an empty string.

### 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 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,
    VkDebugMarkerObjectTagInfoEXT*              pTagInfo);
```

- `device` is the device that created the object.
- `pTagInfo` is a pointer to an instance of the VkDebugMarkerObjectTagInfoEXT structure specifying the parameters of the tag to attach to the object.

### Valid Usage

- `pTagInfo.object` **must** be a Vulkan object
- `pTagInfo.tagName` **must** not be `0`

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pTagInfo` **must** be a pointer to a `VkDebugMarkerObjectTagInfoEXT` structure 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 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 (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 pointer to an array of `tagSize` bytes
- `tagSize` **must** be greater than `0`

### 33.1.2. 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                             commandBuffer,
    VkDebugMarkerMarkerInfoEXT*                 pMarkerInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pMarkerInfo` is a pointer to an instance of the 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 pointer to a `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<br>Secondary | Both | Graphics<br>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 that contains 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 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<br>Secondary | Both | Graphics<br>compute | |

A single marker label can be inserted into a command buffer by calling:

```
void vkCmdDebugMarkerInsertEXT(
    VkCommandBuffer                             commandBuffer,
    VkDebugMarkerMarkerInfoEXT*                 pMarkerInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pMarkerInfo` is a pointer to an instance of the VkDebugMarkerMarkerInfoEXT structure specifying the parameters of the marker to insert.

### Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pMarkerInfo` **must** be a pointer to a `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<br>Secondary | Both | Graphics<br>compute | |

## 33.2. 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` points to a VkDebugReportCallbackCreateInfoEXT structure which defines 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 record the `VkDebugReportCallbackEXT` object created.

### Valid Usage (Implicit)

- `instance` **must** be a valid `VkInstance` handle
- `pCreateInfo` **must** be a pointer to a valid `VkDebugReportCallbackCreateInfoEXT` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a pointer to a valid `VkAllocationCallbacks` structure
- `pCallback` **must** be a 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` indicate which event(s) will cause this callback to be called. Flags are interpreted as bitmasks and multiple **can** be set. Bits which **can** be set include:

  ```
  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,
  } VkDebugReportFlagBitsEXT;
  ```

  - `VK_DEBUG_REPORT_ERROR_BIT_EXT` indicates an error that may cause undefined results, including an application crash.
  - `VK_DEBUG_REPORT_WARNING_BIT_EXT` indicates 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` indicates 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` indicates an informational message such as resource details that may be handy when debugging an application.
  - `VK_DEBUG_REPORT_DEBUG_BIT_EXT` indicates diagnostic information from the implementation and layers.

- `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

- `pfnCallback` **must** be a valid PFN_vkDebugReportCallbackEXT

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of VkDebugReportFlagBitsEXT values

The prototype for the callback function implemented by the application is:

```
typedef VkBool32 (VKAPI_PTR *PFN_vkDebugReportCallbackEXT)(
    VkDebugReportFlagsEXT                       flags,
    VkDebugReportObjectTypeEXT                  objectType,
    uint64_t                                    object,
    size_t                                      location,
    int32_t                                     messageCode,
    const char*                                 pLayerPrefix,
    const char*                                 pMessage,
    void*                                       pUserData);
```

- `flags` indicates the `VkDebugReportFlagBitsEXT` that triggered this callback.
- `objectType` is a VkDebugReportObjectTypeEXT 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 that indicates 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 that indicates to the calling layer the application's desire to abort the call. A value of VK_TRUE indicates that the application wants to abort this call. If the application returns VK_FALSE, the command **must** not be aborted. Applications **should** always return VK_FALSE so that they see the same behavior with and without validation layers enabled.

If the application returns VK_TRUE from its callback and the Vulkan call being aborted returns a VkResult, the layer will return VK_ERROR_VALIDATION_FAILED_EXT.

The object types 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_DESCRIPTOR_UPDATE_TEMPLATE_KHR_EXT = 1000085000,
} VkDebugReportObjectTypeEXT;
```

- VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT is an unknown object.
- VK_DEBUG_REPORT_OBJECT_TYPE_INSTANCE_EXT is a VkInstance.
- VK_DEBUG_REPORT_OBJECT_TYPE_PHYSICAL_DEVICE_EXT is a VkPhysicalDevice.
- VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT is a VkDevice.
- VK_DEBUG_REPORT_OBJECT_TYPE_QUEUE_EXT is a VkQueue.
- VK_DEBUG_REPORT_OBJECT_TYPE_SEMAPHORE_EXT is a VkSemaphore.
- VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_BUFFER_EXT is a VkCommandBuffer.
- VK_DEBUG_REPORT_OBJECT_TYPE_FENCE_EXT is a VkFence.
- VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_MEMORY_EXT is a VkDeviceMemory.
- VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT is a VkBuffer.
- VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_EXT is a VkImage.
- VK_DEBUG_REPORT_OBJECT_TYPE_EVENT_EXT is a VkEvent.
- VK_DEBUG_REPORT_OBJECT_TYPE_QUERY_POOL_EXT is a VkQueryPool.
- VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_VIEW_EXT is a VkBufferView.
- VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_VIEW_EXT is a VkImageView.
- VK_DEBUG_REPORT_OBJECT_TYPE_SHADER_MODULE_EXT is a VkShaderModule.
- VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_CACHE_EXT is a VkPipelineCache.
- VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_LAYOUT_EXT is a VkPipelineLayout.
- VK_DEBUG_REPORT_OBJECT_TYPE_RENDER_PASS_EXT is a VkRenderPass.
- VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_EXT is a VkPipeline.
- VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT_EXT is a VkDescriptorSetLayout.
- VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_EXT is a VkSampler.
- VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_POOL_EXT is a VkDescriptorPool.
- VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_EXT is a VkDescriptorSet.
- VK_DEBUG_REPORT_OBJECT_TYPE_FRAMEBUFFER_EXT is a VkFramebuffer.
- VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_POOL_EXT is a VkCommandPool.
- VK_DEBUG_REPORT_OBJECT_TYPE_SURFACE_KHR_EXT is a VkSurfaceKHR.
- VK_DEBUG_REPORT_OBJECT_TYPE_SWAPCHAIN_KHR_EXT is a VkSwapchainKHR.
- VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT_EXT is a VkDebugReportCallbackEXT.
- VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_KHR_EXT is a VkDisplayKHR.
- VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_MODE_KHR_EXT is a VkDisplayModeKHR.
- VK_DEBUG_REPORT_OBJECT_TYPE_OBJECT_TABLE_NVX_EXT is a VkObjectTableNVX.
- VK_DEBUG_REPORT_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NVX_EXT is a VkIndirectCommandsLayoutNVX.
- VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR_EXT is a VkDescriptorUpdateTemplateKHR.

*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 indicates 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

### 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 string
- `pMessage` **must** be a null-terminated 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 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

# 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.

## Required Versions and Formats

A Vulkan 1.0 implementation **must** support the 1.0 version of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for GLSL.

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

Implementations **must** support the following capability operands declared by `OpCapability`:

- `Matrix`
- `Shader`
- `InputAttachment`
- `Sampled1D`
- `Image1D`
- `SampledBuffer`
- `ImageBuffer`
- `ImageQuery`
- `DerivativeControl`

Implementations **may** support features that are not **required** by the Specification, as described in the Features chapter. If such a feature is supported, then any capability operand(s) corresponding to that feature **must** also be supported.

*Table 59. SPIR-V Capabilities which are not* **required***, and corresponding feature or extension names*

| SPIR-V OpCapability | Vulkan feature or extension name |
|---|---|
| `Geometry` | geometryShader |
| `Tessellation` | tessellationShader |
| `Float64` | shaderFloat64 |
| `Int64` | shaderInt64 |
| `Int16` | shaderInt16 |
| `TessellationPointSize` | shaderTessellationAndGeometryPointSize |
| `GeometryPointSize` | shaderTessellationAndGeometryPointSize |
| `ImageGatherExtended` | shaderImageGatherExtended |

| 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 | shaderStorageImageExtendedFormats |
| InterpolationFunction | sampleRateShading |
| StorageImageReadWithoutFormat | shaderStorageImageReadWithoutFormat |
| StorageImageWriteWithoutFormat | shaderStorageImageWriteWithoutFormat |
| MultiViewport | multiViewport |
| DrawParameters | VK_KHR_shader_draw_parameters |
| SubgroupBallotKHR | VK_EXT_shader_subgroup_ballot |
| SubgroupVoteKHR | VK_EXT_shader_subgroup_vote |
| MultiView | VK_KHX_multiview |
| DeviceGroup | VK_KHX_device_group |
| ImageGatherBiasLodAMD | VK_AMD_texture_gather_bias_lod |
| SampleMaskOverrideCoverageNV | VK_NV_sample_mask_override_coverage |
| GeometryShaderPassthroughNV | VK_NV_geometry_shader_passthrough |
| ShaderViewportIndexLayerNV | VK_NV_viewport_array2 |
| ShaderViewportMaskNV | VK_NV_viewport_array2 |
| PerViewAttributesNV | VK_NVX_multiview_per_view_attributes |

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_shader_ballot 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_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_viewport_array2 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 **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**
  - **Workgroup**
  - **Invocation**
- Variables declared in the **UniformConstant** storage class **must** not have initializers.
- 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.
- Images
  - `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.)
  - `OpSampledImage` **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.
- Decorations
  - 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.
- `OpTypeRuntimeArray` **must** only be used for the last member of an `OpTypeStruct` that is in the `Uniform` storage class and is decorated as `BufferBlock`.
- Linkage: See Shader Interfaces for additional linking and validation rules.
- Compute Shaders
  - For each compute shader entry point, either a `LocalSize` execution mode or an object decorated with the `WorkgroupSize` decoration **must** be specified.
- Atomic instructions **must** declare a scalar 32-bit integer type for the "Result Type".

# Precision and Operation of SPIR-V Instructions

The following rules apply to both 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.
- Any denormalized value input into a shader or potentially generated by any instruction in a shader **may** be flushed to 0.
- The rounding mode **cannot** be set and is undefined.
- NaNs **may** not be generated. Instructions that operate on a NaN **may** not result in a NaN.
- Support for signaling NaNs is **optional** and exceptions are never raised.

The precision of double-precision instructions is at least that of single precision. For single precision (32 bit) instructions, precisions are **required** to be at least as follows, unless decorated with RelaxedPrecision:

*Table 60. Precision of core SPIR-V Instructions*

| Instruction | Precision |
|---|---|
| `OpFAdd` | Correctly rounded. |
| `OpFSub` | Correctly rounded. |
| `OpFMul` | Correctly rounded. |
| `OpFOrdEqual`, `OpFUnordEqual` | Correct result. |
| `OpFOrdLessThan`, `OpFUnordLessThan` | Correct result. |
| `OpFOrdGreaterThan`, `OpFUnordGreaterThan` | Correct result. |
| `OpFOrdLessThanEqual`, `OpFUnordLessThanEqual` | Correct result. |
| `OpFOrdGreaterThanEqual`, `OpFUnordGreaterThanEqual` | Correct result. |
| `OpFDiv` | 2.5 ULP for b in the range $[2^{-126}, 2^{126}]$. |
| conversions between types | Correctly rounded. |

*Table 61. Precision of GLSL.std.450 Instructions*

| Instruction | Precision |
|---|---|
| `fma`() | Inherited from `OpFMul` followed by `OpFAdd`. |
| `exp`(x), `exp2`(x) | $3 + 2 \times \lvert x \rvert$ ULP. |
| `log`(), `log2`() | 3 ULP outside the range [0.5, 2.0]. Absolute error < $2^{-21}$ inside the range [0.5, 2.0]. |
| `pow`(x, y) | Inherited from `exp2`(y × `log2`(x)). |
| `sqrt`() | Inherited from 1.0 / `inversesqrt`(). |
| `inversesqrt`() | 2 ULP. |

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. These include, for example, the trigonometric functions and determinant.

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.

*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 compatibile formats are:

*Table 62. SPIR-V and Vulkan Image Format Compatibility*

| SPIR-V Image Format | Compatible Vulkan Format |
|---|---|
| `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` |

| SPIR-V Image Format | Compatible Vulkan Format |
|---|---|
| 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: 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 63. 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 64. 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 65. Mapping of Vulkan ASTC formats to descriptions*

| VkFormat | Compressed texel block dimensions | sRGB-encoded |
|---|---|---|
| VK_FORMAT_ASTC_4x4_UNORM_BLOCK | 4 × 4 | No |
| VK_FORMAT_ASTC_4x4_SRGB_BLOCK | 4 × 4 | Yes |
| VK_FORMAT_ASTC_5x4_UNORM_BLOCK | 5 × 4 | No |
| VK_FORMAT_ASTC_5x4_SRGB_BLOCK | 5 × 4 | Yes |
| VK_FORMAT_ASTC_5x5_UNORM_BLOCK | 5 × 5 | No |
| VK_FORMAT_ASTC_5x5_SRGB_BLOCK | 5 × 5 | Yes |
| VK_FORMAT_ASTC_6x5_UNORM_BLOCK | 6 × 5 | No |
| VK_FORMAT_ASTC_6x5_SRGB_BLOCK | 6 × 5 | Yes |
| VK_FORMAT_ASTC_6x6_UNORM_BLOCK | 6 × 6 | No |
| VK_FORMAT_ASTC_6x6_SRGB_BLOCK | 6 × 6 | Yes |
| VK_FORMAT_ASTC_8x5_UNORM_BLOCK | 8 × 5 | No |
| VK_FORMAT_ASTC_8x5_SRGB_BLOCK | 8 × 5 | Yes |
| VK_FORMAT_ASTC_8x6_UNORM_BLOCK | 8 × 6 | No |
| VK_FORMAT_ASTC_8x6_SRGB_BLOCK | 8 × 6 | Yes |
| VK_FORMAT_ASTC_8x8_UNORM_BLOCK | 8 × 8 | No |
| VK_FORMAT_ASTC_8x8_SRGB_BLOCK | 8 × 8 | Yes |
| VK_FORMAT_ASTC_10x5_UNORM_BLOCK | 10 × 5 | No |
| VK_FORMAT_ASTC_10x5_SRGB_BLOCK | 10 × 5 | Yes |
| VK_FORMAT_ASTC_10x6_UNORM_BLOCK | 10 × 6 | No |
| VK_FORMAT_ASTC_10x6_SRGB_BLOCK | 10 × 6 | Yes |
| VK_FORMAT_ASTC_10x8_UNORM_BLOCK | 10 × 8 | No |
| VK_FORMAT_ASTC_10x8_SRGB_BLOCK | 10 × 8 | Yes |
| VK_FORMAT_ASTC_10x10_UNORM_BLOCK | 10 × 10 | No |
| VK_FORMAT_ASTC_10x10_SRGB_BLOCK | 10 × 10 | Yes |
| VK_FORMAT_ASTC_12x10_UNORM_BLOCK | 12 × 10 | No |
| VK_FORMAT_ASTC_12x10_SRGB_BLOCK | 12 × 10 | Yes |
| VK_FORMAT_ASTC_12x12_UNORM_BLOCK | 12 × 12 | No |
| VK_FORMAT_ASTC_12x12_SRGB_BLOCK | 12 × 12 | Yes |

# Appendix C: Layers & Extensions

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

http://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`, Khronos `KHX`, multivendor `EXT`, and then alphabetically by author ID. Within each group, extensions are listed in alphabetical order by their name.

> *Note*
>
> The `KHX` author ID indicates that an extension is experimental, and is being considered for standardization in future `KHR` or core Vulkan API versions. Developers are encouraged to experiment with them and provide feedback, but **should** not use them as the basis for production applications. `KHX` extensions are expected to be supported for a limited time only. They **may** change their interfaces and behavior in significant ways as a result of feedback, and **may** be withdrawn or replaced with stable `KHR` or core functionality at any time. Implementations of these extensions receive limited or no testing when submitted to the Khronos conformance process.

## VK_KHR_android_surface

**Name String**

VK_KHR_android_surface

**Extension Type**

Instance extension

**Registered Extension Number**

9

**Last Modified Date**

2016-01-14

**Revision**

6

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**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

**Contacts**

- Jesse Hall, Google

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 GPU 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_descriptor_update_template

**Name String**

VK_KHR_descriptor_update_template

**Extension Type**

Device extension

**Registered Extension Number**

86

**Status**

In Work

**Last Modified Date**

2016-01-11

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Interacts with VK_KHR_push_descriptor

**Contributors**

- Jeff Bolz, NVIDIA
- Michael Worcester, Imagination Technologies

**Contact**

- Markus Tavenrath (matavenrath 'at' nvidia.com)

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's required to rebuild descriptor sets for each frame. For those cases it's 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 which describes 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

### Examples

```
    // TODO: Write some sample code here.
```

### Version History

- Revision 1, 2016-01-11 (Markus Tavenrath)
  - Initial draft

## VK_KHR_display

**Name String**

VK_KHR_display

**Extension Type**

Instance extension

**Registered Extension Number**

3

**Status**

Draft.

**Last Modified Date**

2017-03-13

**Revision**

23

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**Contributors**

- James Jones, NVIDIA
- Norbert Nopper, Freescale
- Jeff Vigil, Qualcomm
- Daniel Rakos, AMD

**Contacts**

- James Jones (jajones 'at' nvidia.com)
- Norbert Nopper (Norbert.Nopper 'at' freescale.com)

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 can not 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 swap chain 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-LoaderAndValidationLayers/blob/master/demos/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 virtualPostion 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 back2 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

**Status**

Draft.

**Last Modified Date**

2017-03-13

**Revision**

10

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_swapchain.
- Requires VK_KHR_display.

**Contributors**

- James Jones, NVIDIA
- Jeff Vigil, Qualcomm
- Jesse Hall, Google

**Contacts**

- James Jones (jajones 'at' nvidia.com)

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/extent 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-LoaderAndValidationLayers/blob/master/demos/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_get_physical_device_properties2

**Name String**

VK_KHR_get_physical_device_properties2

**Extension Type**

Instance extension

**Registered Extension Number**

60

**Status**

Complete.

**Last Modified Date**

2016-11-02

**Revision**

2

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Jeff Bolz, NVIDIA
- Ian Elliott, Google

**Contacts**

- Jeff Bolz (jbolz 'at' nvidia.com)

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

## 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_get_surface_capabilities2

**Name String**

VK_KHR_get_surface_capabilities2

**Extension Type**

Instance extension

**Registered Extension Number**

120

**Last Modified Date**

2017-02-27

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_surface.

**Contributors**

- Ian Elliott, Google
- James Jones, NVIDIA
- Alon Or-bach, Samsung

**Contacts**

- James Jones, NVIDIA

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, combined with VK_KHR_get_display_properties2

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.
- vkGetPhysicalDeviceMirPresentationSupportKHR: Not in this extension.
- vkGetPhysicalDeviceWin32PresentationSupportKHR: Not in this extension.

### Version History

- Revision 1, 2017-02-27 (James Jones)
  - Initial draft.

# VK_KHR_incremental_present

**Name String**

VK_KHR_incremental_present

**Extension Type**

Device extension

**Registered Extension Number**

85

**Last Modified Date**

2016-11-02

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_swapchain (revision 68).

**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

**Contacts**

- Ian Elliott, Google

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 haven't 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 steroescopic-3D swapchains? We need to add a layer for each rectangle. One approach is to create another struct that contains 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_maintenance1

**Name String**

VK_KHR_maintenance1

**Extension Type**

Device extension

**Registered Extension Number**

70

**Status**

Draft

**Last Modified Date**

2016-10-26

**Revision**

1

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**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

**Contacts**

- Piers Daniell (pdaniell 'at' nvidia.com)

**Overview**

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 slink::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

### Issues

None.

### Version History

- Revision 1, 2016-10-26 (Piers Daniell)
  - Internal revisions

## VK_KHR_mir_surface

**Name String**

VK_KHR_mir_surface

**Extension Type**

Instance extension

**Registered Extension Number**

8

**Last Modified Date**

2015-11-28

**Revision**

4

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**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

**Contacts**

- Jesse Hall, Google
- Ian Elliott, LunarG

The VK_KHR_mir_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 Mir surface, 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_MIR_SURFACE_CREATE_INFO_KHR`

### New Enums

None

### New Structures

- VkMirSurfaceCreateInfoKHR

### New Functions

- vkCreateMirSurfaceKHR
- vkGetPhysicalDeviceMirPresentationSupportKHR

### Issues

1) Does Mir need a way to query for compatibility between a particular physical device (and queue family?) and a specific Mir connection, screen, window, etc.?

**RESOLVED**: Yes, vkGetPhysicalDeviceMirPresentationSupportKHR was added to address this.

### 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 vkGetPhysicalDeviceMirPresentationSupportKHR to resolve issue #1.
- Revision 3, 2015-10-26 (Ian Elliott)
  - Renamed from VK_EXT_KHR_mir_surface to VK_KHR_mir_surface.
- Revision 4, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a pCreateInfo structure.

## VK_KHR_push_descriptor

**Name String**

VK_KHR_push_descriptor

**Extension Type**

Device extension

**Registered Extension Number**

81

**Status**

Draft.

**Last Modified Date**

2016-10-15

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_get_physical_device_properties2.

**Contributors**

- Jeff Bolz, NVIDIA
- Michael Worcester, Imagination Technologies

**Contacts**

- Jeff Bolz (jbolz 'at' nvidia.com)

This extension allows descriptors to be written into the command buffer, with the implementation being 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`

## New Enums

None.

### New Structures

- VkPhysicalDevicePushDescriptorPropertiesKHR

### New Functions

- vkCmdPushDescriptorSetKHR

### Issues

None.

### Examples

None.

### Version History

- Revision 1, 2016-10-15 (Jeff Bolz)
  - Internal revisions

## VK_KHR_sampler_mirror_clamp_to_edge

**Name String**

VK_KHR_sampler_mirror_clamp_to_edge

**Extension Type**

Device extension

**Registered Extension Number**

15

**Status**

Final

**Last Modified Date**

2016-02-16

**Revision**

1

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Tobias Hector, Imagination Technologies

**Contacts**

- Tobias Hector (tobias.hector@imgtec.com)

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);
```

## Version History

- Revision 1, 2016-02-16 (Tobias Hector)
  - Initial draft

# VK_KHR_shader_draw_parameters

**Name String**

VK_KHR_shader_draw_parameters

**Extension Type**

Device extension

**Registered Extension Number**

64

**Status**

Complete

**Last Modified Data**

2016-10-05

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires Vulkan 1.0.
- Requires the SPV_KHR_shader_draw_parameters SPIR-V extension.
- Requires GL_ARB_shader_draw_parameters for GLSL source languages.

**Contributors**

- Daniel Koch, NVIDIA Corporation
- Jeff Bolz, NVIDIA
- Daniel Rakos, AMD
- Jan-Harald Fredriksen, ARM
- John Kessenich, Google
- Stuart Smith, IMG

**Contact**

- Daniel Koch (dkoch 'at' nvidia.com)

**Overview**

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

## 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_shared_presentable_image

**Name String**

VK_KHR_shared_presentable_image

**Extension Type**

Device extension

**Registered Extension Number**

112

**Last Modified Date**

2017-03-20

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_swapchain.
- This extension requires VK_KHR_surface.
- This extension requires VK_KHR_get_physical_device_properties2.
- This extension requires VK_KHR_get_surface_capabilities2.

**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

**Contacts**

- Alon Or-bach, Samsung Electronics

This extension extends VK_KHR_swapchain to enable creation of a shared presentable image. This allows the application to use the image while the presention 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 intial 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 CONTINUOUS_REFRESH 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 OUT_OF_DATE 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_surface

**Name String**

VK_KHR_surface

**Extension Type**

Instance extension

**Registered Extension Number**

1

**Last Modified Date**

2016-08-25

**Revision**

25

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**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

**Contacts**

- James Jones, NVIDIA
- Ian Elliott, LunarG

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 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-LoaderAndValidationLayers/blob/master/demos/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 eases 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_swapchain

**Name String**

VK_KHR_swapchain

**Extension Type**

Device extension

**Registered Extension Number**

2

**Last Modified Date**

2016-05-04

**Revision**

68

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_surface.

**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

**Contacts**

- James Jones, NVIDIA
- Ian Elliott, LunarG

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`
- Extending `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`
- Extending `VkResult`:
  - `VK_SUBOPTIMAL_KHR`
  - `VK_ERROR_OUT_OF_DATE_KHR`

## New Enums

None

## New Structures

- VkSwapchainCreateInfoKHR
- VkPresentInfoKHR

## New Functions

- vkCreateSwapchainKHR
- vkDestroySwapchainKHR
- vkGetSwapchainImagesKHR
- vkAcquireNextImageKHR
- vkQueuePresentKHR

## 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 imageArraySize 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 QueuePresent 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_ROT90_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 supportedTransform 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`.

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`.

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 swap chain 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 INHERIT 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 some how, and ensure no other such references exist. Surfaces were not intended to be that heavy-weight.

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-LoaderAndValidationLayers/blob/master/demos/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 rotatation 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 param 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 param 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 CPU 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.

# VK_KHR_wayland_surface

**Name String**

VK_KHR_wayland_surface

**Extension Type**

Instance extension

**Registered Extension Number**

7

**Last Modified Date**

2015-11-28

**Revision**

6

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_surface.

**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

**Contacts**

- Jesse Hall, Google
- Ian Elliott, 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 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_surface

**Name String**

VK_KHR_win32_surface

**Extension Type**

Instance extension

**Registered Extension Number**

10

**Last Modified Date**

2015-11-28

**Revision**

5

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_surface.

**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

**Contacts**

- Jesse Hall, Google
- Ian Elliott, 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.

## 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.

# VK_KHR_xcb_surface

**Name String**

VK_KHR_xcb_surface

**Extension Type**

Instance extension

**Registered Extension Number**

6

**Last Modified Date**

2015-11-28

**Revision**

6

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**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

**Contacts**

- Jesse Hall, Google
- Ian Elliott, 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 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

**Last Modified Date**

2015-11-28

**Revision**

6

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**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

**Contacts**

- Jesse Hall, Google
- Ian Elliott, 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 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_KHX_device_group

**Name String**

VK_KHX_device_group

**Extension Type**

Device extension

**Registered Extension Number**

61

**Status**

Draft.

**Last Modified Date**

2016-10-19

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_swapchain revision 68.

**Contributors**

- Jeff Bolz, NVIDIA

**Contacts**

- Jeff Bolz (jbolz 'at' nvidia.com)

This extension provides functionality to use a logical device that consists of multiple physical devices, as created with the VK_KHX_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_KHX`
  - `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO_KHX`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO_KHX`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO_KHX`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO_KHX`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO_KHX`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO_KHX`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHX`
  - `VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHX`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHX`
  - `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHX`

  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHX`
  - `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHX`
- Extending `VkImageCreateFlagBits`
  - `VK_IMAGE_CREATE_BIND_SFR_BIT_KHX`
- Extending `VkPipelineCreateFlagBits`
  - `VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT_KHX`
  - `VK_PIPELINE_CREATE_DISPATCH_BASE_KHX`
- Extending `VkDependencyFlagBits`
  - `VK_DEPENDENCY_DEVICE_GROUP_BIT_KHX`
- Extending `VkSwapchainCreateFlagBitsKHR`
  - `VK_SWAPCHAIN_CREATE_BIND_SFR_BIT_KHX`

## New Enums

- VkPeerMemoryFeatureFlagBitsKHX
- VkMemoryAllocateFlagBitsKHX
- VkDeviceGroupPresentModeFlagBitsKHX

## New Structures

- VkMemoryAllocateFlagsInfoKHX
- VkBindBufferMemoryInfoKHX
- VkBindImageMemoryInfoKHX
- VkDeviceGroupRenderPassBeginInfoKHX
- VkDeviceGroupCommandBufferBeginInfoKHX
- VkDeviceGroupSubmitInfoKHX
- VkDeviceGroupBindSparseInfoKHX
- VkDeviceGroupPresentCapabilitiesKHX
- VkImageSwapchainCreateInfoKHX
- VkBindImageMemorySwapchainInfoKHX
- VkAcquireNextImageInfoKHX
- VkDeviceGroupPresentInfoKHX
- VkDeviceGroupSwapchainCreateInfoKHX

## New Functions

- vkGetDeviceGroupPeerMemoryFeaturesKHX
- vkBindBufferMemory2KHX
- vkBindImageMemory2KHX

- vkCmdSetDeviceMaskKHX
- vkGetDeviceGroupPresentCapabilitiesKHX
- vkGetDeviceGroupSurfacePresentModesKHX
- vkAcquireNextImage2KHX
- vkCmdDispatchBaseKHX
- vkGetPhysicalDevicePresentRectanglesKHX

### New Built-In Variables

- `DeviceIndex`

### New SPIR-V Capabilities

- `DeviceGroup`

### Issues

None.

### Examples

TODO

### Version History

- Revision 1, 2016-10-19 (Jeff Bolz)
  - Internal revisions

## VK_KHX_device_group_creation

**Name String**

VK_KHX_device_group_creation

**Extension Type**

Instance extension

**Registered Extension Number**

71

**Status**

Draft.

**Last Modified Date**

2016-10-19

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Jeff Bolz, NVIDIA

**Contacts**

- Jeff Bolz (jbolz 'at' nvidia.com)

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_KHX_device_group extension.

## New Object Types

None.

## New Enum Constants

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHX`
- Extending `VkMemoryHeapFlagBits`
  - `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHX`

## New Enums

None.

## New Structures

- VkPhysicalDeviceGroupPropertiesKHX
- VkDeviceGroupDeviceCreateInfoKHX

## New Functions

- vkEnumeratePhysicalDeviceGroupsKHX

## Issues

None.

## Examples

```
    VkDeviceCreateInfo devCreateInfo = { VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO };
    // (not shown) fill out devCreateInfo as usual.
    uint32_t deviceGroupCount = 0;
    VkPhysicalDeviceGroupPropertiesKHX *props = NULL;

    // Query the number of device groups
    vkEnumeratePhysicalDeviceGroupsKHX(g_vkInstance, &deviceGroupCount, NULL);

    // Allocate and initialize structures to query the device groups
    props = (VkPhysicalDeviceGroupPropertiesKHX *)malloc(deviceGroupCount*sizeof
(VkPhysicalDeviceGroupPropertiesKHX));
    for (i = 0; i < deviceGroupCount; ++i) {
        props[i].sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHX;
        props[i].pNext = NULL;
    }
    vkEnumeratePhysicalDeviceGroupsKHX(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.
    VkDeviceGroupDeviceCreateInfoKHX deviceGroupInfo = {
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO_KHX };
    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_KHX_external_memory_capabilities

### Name String

VK_KHX_external_memory_capabilities

### Extension Type

Instance extension

### Registered Extension Number

72

**Status**

Draft

**Last Modified Date**

2016-10-17

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_get_physical_device_properties2.
- Interacts with VK_NV_dedicated_allocation.

**Contributors**

- Ian Elliot, Google
- Jesse Hall, Google
- James Jones, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

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_KHX`
- `VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES_KHX`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO_KHX`
- `VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES_KHX`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHX`
- `VK_LUID_SIZE_KHX`

## New Enums

- VkExternalMemoryHandleTypeFlagBitsKHX

- VkExternalMemoryFeatureFlagBitsKHX

## New Structs

- VkExternalMemoryPropertiesKHX
- VkPhysicalDeviceExternalImageFormatInfoKHX
- VkExternalImageFormatPropertiesKHX
- VkPhysicalDeviceExternalBufferInfoKHX
- VkExternalBufferPropertiesKHX
- VkPhysicalDeviceIDPropertiesKHX

## New Functions

- vkGetPhysicalDeviceExternalBufferPropertiesKHX

## 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 can not 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 VkExternalImageFormatPropertiesKHX and VkExternalBufferPropertiesKHX 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 midly 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.

# VK_KHX_external_memory

**Name String**

VK_KHX_external_memory

**Extension Type**

Device extension

**Registered Extension Number**

73

**Status**

Draft

**Last Modified Date**

2016-10-20

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_external_memory_capabilities.
- Interacts with VK_NV_dedicated_allocation.

**Contributors**

- Ian Elliot, Google

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA
- Tobias Hector, Imagination

**Contact**

James Jones (jajones 'at' nvidia.com)

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_KHX`
- `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_KHX`
- `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHX`
- `VK_QUEUE_FAMILY_EXTERNAL_KHX`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX`

## New Enums

None.

## New Structs

- VkExternalMemoryImageCreateInfoKHX
- VkExternalMemoryBufferCreateInfoKHX
- VkExportMemoryAllocateInfoKHX

## New Functions

None.

## 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_KHX_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 to 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 accomodate 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 interopability 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 obects 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 can not be used to complete the requested import operation. However, implementations need not validate handles are of the exact type specified by the application.

# VK_KHX_external_memory_win32

**Name String**

VK_KHX_external_memory_win32

**Extension Type**

Device extension

**Registered Extension Number**

74

**Status**

Draft

**Last Modified Date**

2016-10-21

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_external_memory.

**Contributors**

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

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_KHX`
- `VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHX`
- `VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHX`

## New Enums

None.

## New Structs

- VkImportMemoryWin32HandleInfoKHX
- VkExportMemoryWin32HandleInfoKHX
- VkMemoryWin32HandlePropertiesKHX

### New Functions

- vkGetMemoryWin32HandleKHX
- vkGetMemoryWin32HandlePropertiesKHX

### Issues

1) Do applications need to call CloseHandle() on the values returned from vkGetMemoryWin32HandleKHX when `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX`?

**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.

# VK_KHX_external_memory_fd

**Name String**

VK_KHX_external_memory_fd

**Extension Type**

Device extension

**Registered Extension Number**

75

**Status**

Draft

**Last Modified Date**

2016-10-21

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_external_memory.

**Contributors**

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

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_KHX`
- `VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHX`

## New Enums

None.

## New Structs

- VkImportMemoryFdInfoKHX
- VkMemoryFdPropertiesKHX

## New Functions

- vkGetMemoryFdKHX
- vkGetMemoryFdPropertiesKHX

## Issues

1) Does the application need to close the file descriptor returned by vkGetMemoryFdKHX?

**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.

# VK_KHX_win32_keyed_mutex

**Name String**

VK_KHX_win32_keyed_mutex

**Extension Type**

Device extension

**Registered Extension Number**

76

**Status**

Draft

**Last Modified Date**

2016-10-21

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_external_memory.

**Contributors**

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

**Contact**

Carsten Rohde (crohde 'at' nvidia.com)

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_KHX`

## New Enums

None.

## New Structs

- VkWin32KeyedMutexAcquireReleaseInfoKHX

## New Functions

None.

## Issues

None.

# VK_KHX_external_semaphore_capabilities

**Name String**

VK_KHX_external_semaphore_capabilities

**Extension Type**

Instance extension

**Registered Extension Number**

77

**Status**

Draft

**Last Modified Date**

2016-10-20

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_get_physical_device_properties2.

**Contributors**

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

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_KHX`
- `VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES_KHX`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHX`
- `VK_LUID_SIZE_KHX`

## New Enums

- VkExternalSemaphoreHandleTypeFlagBitsKHX
- VkExternalSemaphoreFeatureFlagBitsKHX

## New Structs

- VkPhysicalDeviceExternalSemaphoreInfoKHX
- VkExternalSemaphorePropertiesKHX
- VkPhysicalDeviceIDPropertiesKHX

## New Functions

- vkGetPhysicalDeviceExternalSemaphorePropertiesKHX

## Issues

# VK_KHX_external_semaphore

**Name String**

VK_KHX_external_semaphore

**Extension Type**

Device extension

**Registered Extension Number**

78

**Status**

Draft

**Last Modified Date**

2016-10-21

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_external_semaphore_capabilities.

**Contributors**

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

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_KHX`
- `VK_ERROR_INVALID_EXTERNAL_HANDLE_KHX`

### New Enums

None.

### New Structs

- VkExportSemaphoreCreateInfoKHX

### New Functions

None.

### 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 undesireable 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 can not be used to complete the requested import operation. However, implementations need not validate handles are of the exact type specified by the application.

## VK_KHX_external_semaphore_win32

**Name String**

VK_KHX_external_semaphore_win32

**Extension Type**

Device extension

**Registered Extension Number**

79

**Status**

Draft

**Last Modified Date**

2016-10-21

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_external_semaphore_capabilities.

**Contributors**

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore state to and import semaphore state from Windows handles.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHX`
- `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHX`
- `VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHX`

## New Enums

None.

## New Structs

- VkImportSemaphoreWin32HandleInfoKHX
- VkExportSemaphoreWin32HandleInfoKHX
- VkD3D12FenceSubmitInfoKHX

### New Functions

- vkImportSemaphoreWin32HandleKHX
- vkGetSemaphoreWin32HandleKHX

### Issues

1) Do applications need to call `CloseHandle`() on the values returned from vkGetSemaphoreWin32HandleKHX when `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX`?

**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.

# VK_KHX_external_semaphore_fd

**Name String**

VK_KHX_external_semaphore_fd

**Extension Type**

Device extension

**Registered Extension Number**

80

**Status**

Draft

**Last Modified Date**

2016-10-21

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_external_semaphore_capabilities.

**Contributors**

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore state to and import semaphore state from POSIX file descriptors.

## New Object Types

None.

## New Enum Constants

- `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHX`

## New Enums

None.

## New Structs

- VkImportSemaphoreFdInfoKHX

### New Functions

- vkImportSemaphoreFdKHX
- vkGetSemaphoreFdKHX

### Issues

1) Does the application need to close the file descriptor returned by vkGetSemaphoreFdKHX?

**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.

# VK_KHX_multiview

**Name String**

VK_KHX_multiview

**Extension Type**

Device extension

**Registered Extension Number**

54

**Status**

Draft.

**Last Modified Date**

2016-10-28

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_get_physical_device_properties2.

**Contributors**

- Jeff Bolz, NVIDIA

**Contacts**

- Jeff Bolz (jbolz 'at' nvidia.com)

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_KHX`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES_KHX`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES_KHX`
- Extending `VkDependencyFlagBits`
  - `VK_DEPENDENCY_VIEW_LOCAL_BIT_KHX`

## New Enums

None.

## New Structures

- VkPhysicalDeviceMultiviewFeaturesKHX
- VkPhysicalDeviceMultiviewPropertiesKHX
- VkRenderPassMultiviewCreateInfoKHX

## New Functions

None.

## New Built-In Variables

- `ViewIndex`

## New SPIR-V Capabilities

- `MultiView`

## Issues

None.

## Examples

None.

## Version History

- Revision 1, 2016-10-28 (Jeff Bolz)
  - Internal revisions

# VK_EXT_debug_marker

**Name String**

VK_EXT_debug_marker

**Extension Type**

Device extension

**Registered Extension Number**

23

**Last Modified Date**

2017-01-31

**Revision**

4

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.11 of the Vulkan API.
- Requires VK_EXT_debug_report.

**Contributors**

- Baldur Karlsson
- Dan Ginsburg, Valve
- Jon Ashburn, LunarG
- Kyle Spagnoli, NVIDIA

**Contacts**

- Baldur Karlsson

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 organisation 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 VkDeviceCreateInfo 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 `Vk*CreateInfo` 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 `Vk*CreateInfo` time does not allow renaming, prevents late binding of naming information, and does not allow naming of implicitly created objects such as queues and swap chain 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

**Registered Extension Number**

12

**Last Modified Date**

2017-04-27

**Revision**

8

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.27 of the Vulkan API.

**Contributors**

- Courtney Goeltzenleuchter, LunarG
- Dan Ginsburg, Valve
- Jon Ashburn, LunarG
- Mark Lobodzinski, LunarG

**Contacts**

- Courtney Goeltzenleuchter

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*

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 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` indicates 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

# VK_EXT_validation_flags

**Name String**

VK_EXT_validation_flags

**Extension Type**

Instance extension

**Registered Extension Number**

62

**Last Modified Date**

2016-09-06

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.25 of the Vulkan API.

**Contributors**

- Tobin Ehlis, Google
- Courtney Goeltzenleuchter, Google

**Contacts**

- Tobin Ehlis, Google (tobine@google.com)

This extension provides the VkValidationFlagsEXT struct that can be included in the `pNext` chain at vkCreateInstance time. The new struct contains an array of VkValidationCheckEXT values that will be disabled by the validation layers.

## 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 1, 2016-08-26 (Courtney Goeltzenleuchter)
  - Initial draft

# VK_EXT_direct_mode_display

**Name String**

VK_EXT_direct_mode_display

**Extension Type**

Instance extension

**Registered Extension Number**

89

**Last Modified Date**

2016-12-13

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.37 of the Vulkan API.
- Requires VK_KHR_display

**Contributors**

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Liam Middlebrook, NVIDIA

**Contacts**

- James Jones, NVIDIA (jajones 'at' nvidia.com)

This is extension, along with related platform exentions, 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_acquire_xlib_display

**Name String**

VK_EXT_acquire_xlib_display

**Extension Type**

Instance extension

**Registered Extension Number**

90

**Last Modified Date**

2016-12-13

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.37 of the Vulkan API.
- Requires VK_KHR_display.
- Requires VK_EXT_direct_mode_display.

**Contributors**

- Dave Airlie, Red Hat
- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Liam Middlebrook, NVIDIA
- Daniel Vetter, Intel

**Contacts**

- James Jones, NVIDIA (jajones 'at' nvidia.com)

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_discard_rectangles

**Name String**

VK_EXT_discard_rectangles

**Extension Type**

Device extension

**Registered Extension Number**

100

**Status**

Complete

**Last Modified Data**

2016-12-22

**Revision**

1

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires Vulkan 1.0.
- Requires VK_KHR_get_physical_device_properties2.

**Contributors**

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

**Contact**

- Piers Daniell (pdaniell 'at' nvidia.com)

**Overview**

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. If the VK_KHX_device_group extension is enabled the discard rectangles can be different for each physical device in the 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

- VkPipelineDiscardRectangleStateCreateFlagsEXT
- VkDiscardRectangleModeEXT

### New Structures

- VkPhysicalDeviceDiscardRectanglePropertiesEXT
- VkPipelineDiscardRectangleStateCreateInfoEXT

### New Functions

- vkCmdSetDiscardRectangleEXT

### Issues

None.

### Version History

- Revision 1, 2016-12-22 (Piers Daniell)
  - Internal revisions

## VK_EXT_display_surface_counter

**Name String**

VK_EXT_display_surface_counter

**Extension Type**

Instance extension

**Registered Extension Number**

91

**Last Modified Date**

2016-12-13

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.37 of the Vulkan API.
- Requires VK_KHR_display

**Contributors**

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Daniel Vetter, Intel

**Contacts**

- James Jones, NVIDIA (jajones 'at' nvidia.com)

This is 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_CAPABILITIES2_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_display_control

**Name String**

VK_EXT_display_control

**Extension Type**

Device extension

**Registered Extension Number**

92

**Last Modified Date**

2016-12-13

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.37 of the Vulkan API.
- Requires VK_KHR_display
- Requires VK_EXT_display_surface_counter
- Requires VK_KHR_swapchain

**Contributors**

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA

- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Daniel Vetter, Intel

**Contacts**

- James Jones, NVIDIA (jajones 'at' nvidia.com)

This extension defines a set of utility functions for use with the VK_KHR_display and VK_KHR_display_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_shader_subgroup_ballot

**Name String**

VK_EXT_shader_subgroup_ballot

**Extension Type**

Device extension

**Registered Extension Number**

65

**Status**

Draft

**Last Modified Date**

2016-11-28

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires Vulkan 1.0.

- 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

**Contact**

- Daniel Koch (dkoch 'at' nvidia.com)

**Overview**

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

**Status**

Draft

**Last Modified Date**

2016-11-28

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires Vulkan 1.0.
- 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

**Contact**

- Daniel Koch (dkoch 'at' nvidia.com)

**Overview**

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_hdr_metadata

**Name String**

VK_EXT_hdr_metadata

**Extension Type**

Device

**Registered Extension Number**

106

**Last Modified Date**

2017-03-04

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_surface.

**Contributors**

- Courtney Goeltzenleuchter, Google

**Contacts**

- 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 Assocation) 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

## VK_EXT_swapchain_colorspace

**Name String**

VK_EXT_swapchain_colorspace

**Extension Type**

Instance

**Registered Extension Number**

105

**Last Modified Date**

2017-03-15

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_surface

**Contributors**

- Courtney Goeltzenleuchter, Google

**Contacts**

- 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 OETF.
  - `VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT` - supports the extended sRGB color space and applies a linear OETF.
  - `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 OETF.
  - `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 colorspaces. 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.

## VK_AMD_draw_indirect_count

**Name String**

VK_AMD_draw_indirect_count

**Extension Type**

Device extension

**Registered Extension Number**

34

**Last Modified Date**

2016-08-23

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Matthaeus G. Chajdas, AMD
- Derrick Owens, AMD
- Graham Sellers, AMD
- Daniel Rakos, AMD
- Dominik Witczak, AMD

**Contacts**

- Matthaeus G. Chajdas, AMD (matthaeus.chajdas@amd.com)

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

## Version History

- Revision 2, 2016-08-23 (Dominik Witczak)
  - Minor fixes
- Revision 1, 2016-07-21 (Matthaeus Chajdas)
  - Initial draft

# VK_AMD_gcn_shader

**Name String**

VK_AMD_gcn_shader

**Extension Type**

Device extension

**Registered Extension Number**

26

**Last Modified Date**

2016-05-30

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.15 of the Vulkan API.

**Contributors**

- Dominik Witczak, AMD
- Daniel Rakos, AMD
- Rex Xu, AMD
- Graham Sellers, AMD

**Contacts**

- Dominik Witczak, AMD (dominik.witczak@amd.com)

This extension adds support for the following SPIR-V extension in {apiname}:

- SPV_AMD_gcn_shader

## Version History

- Revision 1, 2016-05-30 (Dominik Witczak)
  - Initial draft

# VK_AMD_gpu_shader_half_float

**Name String**

VK_AMD_gpu_shader_half_float

**Extension Type**

Device extension

**Registered Extension Number**

37

**Last Modified Date**

2016-09-21

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.27 of the Vulkan API.

**Contributors**

- Daniel Rakos, AMD
- Dominik Witczak, AMD
- Donglin Wei, AMD
- Graham Sellers, AMD
- Qun Lin, AMD
- Rex Xu, AMD

**Contacts**

- Dominik Witczak, AMD (Dominik.Witczak@amd.com)

This extension adds support for the following SPIR-V extension in {apiname}:

- SPV_AMD_gpu_shader_half_float

## Version History

- Revision 1, 2019-09-21 (Dominik Witczak)
  - Initial draft

# VK_AMD_negative_viewport_height

**Name String**

VK_AMD_negative_viewport_height

**Extension Type**

Device extension

**Registered Extension Number**

36

**Last Modified Date**

2016-09-02

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Matthaeus G. Chajdas, AMD

- Graham Sellers, AMD
- Baldur Karlsson

**Contacts**

- Matthaeus G. Chajdas, AMD (matthaeus.chajdas@amd.com)

This extension allows an application to specify a negative viewport height. The result is that the viewport transformation will flip along the y-axis.

- Revision 1, 2016-09-02 (Matthaeus Chajdas)
  - Initial draft

# VK_AMD_rasterization_order

**Name String**

VK_AMD_rasterization_order

**Extension Type**

Device extension

**Registered Extension Number**

19

**Last Modified Date**

2016-04-25

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.11 of the Vulkan API.

**Contributors**

- Matthaeus G. Chajdas, AMD
- Jaakko Konttinen, AMD
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Dominik Witczak, AMD

**Contacts**

- Daniel Rakos, 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. The rasterization order is completely implementation dependent in this case 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

**Last Modified Date**

2016-09-19

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.27 of the Vulkan API.

**Contributors**

- Qun Lin, AMD
- Graham Sellers, AMD
- Daniel Rakos, AMD
- Rex Xu, AMD
- Dominik Witczak, AMD
- Matthäus G. Chajdas, AMD

**Contacts**

- Dominik Witczak, AMD (Dominik.Witczak@amd.com)

This extension adds support for the following SPIR-V extension in {apiname}:

- SPV_AMD_shader_ballot

## Version History

- Revision 1, 2016-09-19 (Dominik Witczak)
  - Initial draft

# VK_AMD_shader_explicit_vertex_parameter

**Name String**

VK_AMD_shader_explicit_vertex_parameter

**Extension Type**

Device extension

**Registered Extension Number**

22

**Last Modified Date**

2016-05-10

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.11 of the Vulkan API.

**Contributors**

- Matthaeus G. Chajdas, AMD
- Qun Lin, AMD
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Rex Xu, AMD

**Contacts**

- Qun Lin, AMD (quentin.lin@amd.com)

This extension adds support for the following SPIR-V extension in {apiname}:

- SPV_AMD_shader_explicit_vertex_parameter

## Version History

- Revision 1, 2016-05-10 (Daniel Rakos)
  - Initial draft

# VK_AMD_shader_trinary_minmax

**Name String**

VK_AMD_shader_trinary_minmax

**Extension Type**

Device extension

**Registered Extension Number**

21

**Last Modified Date**

2016-05-10

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0.11 of the Vulkan API.

**Contributors**

- Matthaeus G. Chajdas, AMD
- Qun Lin, AMD
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Rex Xu, AMD

**Contacts**

- Qun Lin, AMD (quentin.lin@amd.com)

This extension adds support for the following SPIR-V extension in {apiname}:

- 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

**Last Modified Date**

03/21/2017

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_get_physical_device_properties2.
- 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

**Contacts**

- Rex Xu, AMD (Rex.Xu@amd.com)

This extension adds two related features.

Firstly, support for the following SPIR-V extension in {apiname} 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.
VkPhysicalDeviceImageFormatInfo2KHR   formatInfo;
VkImageFormatProperties2KHR           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_KHR;
formatInfo.pNext  = nullptr;
formatInfo.format = ...;
formatInfo.type   = ...;
formatInfo.tiling = ...;
formatInfo.usage  = ...;
formatInfo.flags  = ...;

formatProps.sType = VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2_KHR;
formatProps.pNext = &textureLODGatherSupport;

vkGetPhysicalDeviceImageFormatProperties2KHR(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_GOOGLE_display_timing

**Name String**

VK_GOOGLE_display_timing

**Extension Type**

Device extension

**Registered Extension Number**

93

**Last Modified Date**

2017-02-14

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires VK_KHR_swapchain.

**Contributors**

- Ian Elliott, Google
- Jesse Hall, Google

**Contacts**

- Ian Elliott, 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 can allow the application's animation to look smooth to the user, with no stuttering.

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 the 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-LoaderAndValidationLayers/blob/master/demos/cube.c).

## Version History

- Revision 1, 2017-02-14 (Ian Elliott)
  - Internal revisions

# VK_IMG_filter_cubic

**Name String**

VK_IMG_filter_cubic

**Extension Type**

Device extension

**Registered Extension Number**

16

**Status**

Final

**Last Modified Date**

2016-02-23

**Revision**

1

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Tobias Hector, Imagination Technologies

**Contacts**

- Tobias Hector (tobias.hector@imgtec.com)

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 hardware.

## New Enum Constants

- Extending VkFilter:
  - `VK_FILTER_CUBIC_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_MVK_ios_surface

**Name String**

VK_MVK_ios_surface

**Extension Type**

Instance extension

**Registered Extension Number**

123

**Last Modified Date**

2017-02-24

**Revision**

2

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**Contributors**

- Bill Hollings, The Brenwill Workshop Ltd.

**Contacts**

- 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

**Last Modified Date**

2017-02-24

**Revision**

2

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**Contributors**

- Bill Hollings, The Brenwill Workshop Ltd.

**Contacts**

- 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

**Last Modified Date**

2016-12-2

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_KHR_surface.

**Contributors**

- Mathias Heyer, NVIDIA
- Michael Chock, NVIDIA
- Yasuhiro Yoshioka, Nintendo
- Daniel Koch, NVIDIA

**Contacts**

- Mathias Heyer, 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_NV_clip_space_w_scaling

**Name String**

VK_NV_clip_space_w_scaling

**Extension Type**

Device extension

**Registered Extension Number**

88

**Status**

Complete.

**Last Modified Date**

2017-02-15

**Revision**

1

**Dependencies**

- This extension is written against version 1.0. of the Vulkan API.

**Contributors**

- Eric Werness, NVIDIA
- Kedarnath Thangudu, NVIDIA

**Contacts**

- Eric Werness (ewerness 'at' nvidia.com)

**Overview**

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 4 viewports, one for each of the 4 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_DYANMIC_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 it's 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];

VkPipelineViewportStateCreateInfo 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_dedicated_allocation

#### Name String

VK_NV_dedicated_allocation

**Extension Type**

Device extension

**Registered Extension Number**

27

**Status**

Draft.

**Last Modified Date**

2016-05-31

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Jeff Bolz, NVIDIA

**Contacts**

- Jeff Bolz (jbolz 'at' nvidia.com)

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 am 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_capabilities

#### Name String

VK_NV_external_memory_capabilities

**Extension Type**

Instance extension

**Registered Extension Number**

56

**Status**

Complete

**Last Modified Date**

2016-08-19

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Interacts with VK_NV_dedicated_allocation.

**Contributors**

- James Jones, NVIDIA

**Contact**

James Jones (jajones 'at' nvidia.com)

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 can not 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 midly weird to have some platform-specific ones defined in the core spec and some in extensions

# VK_NV_external_memory

**Name String**

VK_NV_external_memory

**Extension Type**

Device extension

**Registered Extension Number**

57

**Status**

Complete

**Last Modified Date**

2016-08-19

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_NV_external_memory_capabilities.

**Contributors**

- James Jones, NVIDIA
- Carsten Rohde, NVIDIA

**Contact**

- James Jones (jajones 'at' nvidia.com)

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 applictions 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 interopability extensions), or Vulkan incompatible. When sharing images with Vulkan incompatible APIs, the Vulkan image must be transitioned to the 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_win32

**Name String**

VK_NV_external_memory_win32

**Extension Type**

Device extension

**Registered Extension Number**

58

**Status**

Complete

**Last Modified Date**

2016-08-19

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- Requires VK_NV_external_memory_capabilities.
- Requires VK_NV_external_memory.

**Contributors**

- James Jones, NVIDIA
- Carsten Rohde, NVIDIA

**Contact**

- James Jones (jajones 'at' nvidia.com)

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 applictions 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 interopability extensions), or Vulkan incompatible. When sharing images with Vulkan incompatible APIs, the Vulkan image must be transitioned to the 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_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHX`?

**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_geometry_shader_passthrough

#### Name String

VK_NV_geometry_shader_passthrough

#### Extension Type

Device extension

#### Registered Extension Number

96

#### Status

Complete

#### Last Modified Data

2017-02-15

**Revision**

1

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires Vulkan 1.0.
- 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

**Contact**

- Daniel Koch (dkoch 'at' nvidia.com)

**Overview**

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 indicates 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 straight forward 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 | layout(points, max_vertices=1) |
   | lines | layout(line_stripo, max_vertices=2) |
   | triangles | layout(triangle_strip, max_vertices=3) |

2. How does interface matching work with passthrough geometry shaders?

   RESOLVED: This is described in Passthrough Interface Matching. In GL when using passthough 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 varables that add the passthrough layout qualifier are exempted from the rule requiring location assignment because built-in variables are 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.

// Redeclare 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_glsl_shader

**Name String**

VK_NV_glsl_shader

**Extension Type**

Device extension

**Registered Extension Number**

13

**Status**

Draft.

**Last Modified Date**

2016-02-14

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Piers Daniell, NVIDIA

**Contacts**

- Piers Daniell (pdaniell 'at' nvidia.com)

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 appropriatly.

## 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_viewport_array2

#### Name String

VK_NV_viewport_array2

#### Extension Type

Device extension

#### Registered Extension Number

97

#### Status

Complete

#### Last Modified Data

2017-02-15

#### Revision

1

#### Dependencies

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires Vulkan 1.0.

- 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

**Contact**

- Daniel Koch (dkoch 'at' nvidia.com)

**Overview**

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 specificiation.

## 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

**Status**

Complete

**Last Modified Data**

2016-12-22

**Revision**

1

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires Vulkan 1.0.
- This extension requires `multiViewport` and `geometryShader` features to be useful.

**Contributors**

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

**Contact**

- Piers Daniell (pdaniell 'at' nvidia.com)

**Overview**

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 (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, NV_viewport_array2, and 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..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..5) is added to the base gl_Layer value of zero 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_eye.

2. On NVIDIA implementations supporting floating-point depth buffers with values outside [0,1], prevent unwanted near plane clipping by enabling DEPTH_CLAMP. 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_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 GREATER instead of LESS.

## Version History

- Revision 1, 2016-12-22 (Piers Daniell)
  - Internal revisions

# VK_NV_win32_keyed_mutex

**Name**

VK_NV_win32_keyed_mutex

**Extension Type**

Device extension

**Registered Extension Number**

59

**Status**

Complete

**Last Modified Date**

2016-08-19

**Revision**

2

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

- Requires VK_NV_external_memory_capabilities
- Requires VK_NV_external_memory_win32

**Contributors**

- James Jones, NVIDIA
- Carsten Rohde, NVIDIA

**Contact**

- Carsten Rohde (crohde 'at' nvidia.com)

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.

# VK_NV_sample_mask_override_coverage

**Name String**

VK_NV_sample_mask_override_coverage

**Extension Type**

Device extension

**Registered Extension Number**

95

**Status**

Complete

**Last Modified Data**

2016-12-08

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- This extension requires Vulkan 1.0.
- 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

**Contact**

- Piers Daniell (pdaniell 'at' nvidia.com)

**Overview**

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_NVX_device_generated_commands

**Name String**

VK_NVX_device_generated_commands

**Extension Type**

Device extension

**Registered Extension Number**

87

**Last Modified Date**

2016-10-31

**Revision**

1

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.

**Contributors**

- Pierre Boudier, NVIDIA
- Christoph Kubisch, NVIDIA
- Mathias Schott, NVIDIA
- Jeff Bolz, NVIDIA
- Eric Werness, NVIDIA
- Detlef Roettger, NVIDIA
- Daniel Koch, NVIDIA

**Contacts**

- Pierre Boudier, NVIDIA (pboudier@nvidia.com)
- Christoph Kubisch, NVIDIA (ckubisch@nvidia.com)

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… 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...).

## 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 ;)

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?

Similar to VkPipeline, signatures have the most likeliness to be cross-vendor adoptable. They also benefit from being processable in parallel.

3) How to name sequence description

ExecuteCommandSignature a bit long, just ExecuteSignature or actually more Vulkan nomenclature IndirectCommandsLayout

4) Do we want to provide indirectCommands inputs with layout or at indirectCommands time?

Separate layout from data as Vulkan does. Provide full flexibilty for indirectCommands.

5) Should the input be provided as SoA or AoS?

It is desired by application to reuse the list of objects and render them with some kind 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 IndirectCommandsLayout'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)
   flink:vkCmdProcessCommandsNVX makes use slink:VkCommandBufferBeginInfo
   and serves as flink:vkBeginCommandBuffer / flink:vkEndCommandBuffer
   implicitly.
c) The targetCommandbuffer must be inside the "`begin`" state already at
   the moment of being passed.
   This very likely suggests a new slink:VkCommandBufferUsageFlags
   etext:VK_COMMAND_BUFFER_USAGE_DEVICE_GENERATED_BIT.
d) The targetCommandbuffer must reserve space via a new function.
```

used a & 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 support 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 DRAW_INDIRECT 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

TODO links to gameworks & designworks samples

```
  // 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 1, 2016-10-31 (Christoph Kubisch)
  - Initial draft

# VK_NVX_multiview_per_view_attributes

#### Name String

VK_NVX_multiview_per_view_attributes

**Extension Type**

Device extension

**Registered Extension Number**

98

**Status**

Draft.

**Last Modified Date**

2017-01-13

**Revision**

1

**IP Status**

No known IP claims.

**Dependencies**

- This extension is written against version 1.0 of the Vulkan API.
- 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.
- Requires VK_KHR_get_physical_device_properties2 and VK_KHX_multiview.
- This extension interacts with VK_NV_viewport_array2.

**Contributors**

- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA

**Contacts**

- Jeff Bolz (jbolz 'at' nvidia.com)

**Overview**

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

# Appendix D: 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.

## Structure Types

Vulkan structures containing `sType` members **must** have a value of `sType` matching the type of the structure, as described more fully in Valid Usage for Structure Types. 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_SWAPCHAIN_CREATE_INFO_KHR = 1000001000,
    VK_STRUCTURE_TYPE_PRESENT_INFO_KHR = 1000001001,
    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_MIR_SURFACE_CREATE_INFO_KHR = 1000007000,
    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_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD = 1000041000,
    VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO_KHX = 1000053000,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES_KHX = 1000053001,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES_KHX = 1000053002,
    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_PHYSICAL_DEVICE_FEATURES_2_KHR = 1000059000,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2_KHR = 1000059001,
    VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2_KHR = 1000059002,
    VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2_KHR = 1000059003,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2_KHR = 1000059004,
    VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2_KHR = 1000059005,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2_KHR = 1000059006,
    VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2_KHR = 1000059007,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2_KHR = 1000059008,
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO_KHX = 1000060000,
    VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO_KHX = 1000060001,
```

```
    VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO_KHX = 1000060002,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO_KHX = 1000060003,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO_KHX = 1000060004,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO_KHX = 1000060005,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO_KHX = 1000060006,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHX = 1000060007,
    VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHX = 1000060008,
    VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHX = 1000060009,
    VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHX = 1000060010,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHX = 1000060011,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHX = 1000060012,
    VK_STRUCTURE_TYPE_VALIDATION_FLAGS_EXT = 1000061000,
    VK_STRUCTURE_TYPE_VI_SURFACE_CREATE_INFO_NN = 1000062000,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHX = 1000070000,
    VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO_KHX = 1000070001,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO_KHX = 1000071000,
    VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES_KHX = 1000071001,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO_KHX = 1000071002,
    VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES_KHX = 1000071003,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHX = 1000071004,
    VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO_KHX = 1000072000,
    VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_KHX = 1000072001,
    VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHX = 1000072002,
    VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHX = 1000073000,
    VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHX = 1000073001,
    VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHX = 1000073002,
    VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHX = 1000074000,
    VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHX = 1000074001,
    VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHX = 1000075000,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO_KHX = 1000076000,
    VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES_KHX = 1000076001,
    VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHX = 1000077000,
    VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHX = 1000078000,
    VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHX = 1000078001,
    VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHX = 1000078002,
    VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHX = 1000079000,
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PUSH_DESCRIPTOR_PROPERTIES_KHR = 1000080000,
    VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR = 1000084000,
    VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO_KHR = 1000085000,
    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_CAPABILITIES2_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_HDR_METADATA_EXT = 1000105000,
    VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR = 1000111000,
    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_IOS_SURFACE_CREATE_INFO_MVK = 1000122000,
    VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK = 1000123000,
} VkStructureType;
```

# Flag Types

Vulkan flag types are all bitmasks aliasing the base type `VkFlags` and with corresponding bit flag types defining the valid bits for that flag, as described in Valid Usage for Flags. Flag types supported by the Vulkan API include:

```
typedef VkFlags VkAccessFlags;
```

```
typedef VkFlags VkAttachmentDescriptionFlags;
```

```
typedef VkFlags VkBufferCreateFlags;
```

```
typedef VkFlags VkBufferUsageFlags;
```

```
typedef VkFlags VkBufferViewCreateFlags;
```

```
typedef VkFlags VkColorComponentFlags;
```

```
typedef VkFlags VkCommandBufferResetFlags;
```

```
typedef VkFlags VkCommandBufferUsageFlags;
```

```
typedef VkFlags VkCommandPoolCreateFlags;
```

```
typedef VkFlags VkCommandPoolResetFlags;
```

```
typedef VkFlags VkCullModeFlags;
```

```
typedef VkFlags VkDependencyFlags;
```

```
typedef VkFlags VkDescriptorPoolCreateFlags;
```

```
typedef VkFlags VkDescriptorPoolResetFlags;
```

```
typedef VkFlags VkDescriptorSetLayoutCreateFlags;
```

```
typedef VkFlags VkDescriptorUpdateTemplateCreateFlagsKHR;
```

```
typedef VkFlags VkDeviceCreateFlags;
```

```
typedef VkFlags VkDeviceGroupPresentModeFlagsKHX;
```

```
typedef VkFlags VkDeviceQueueCreateFlags;
```

```
typedef VkFlags VkEventCreateFlags;
```

```
typedef VkFlags VkFenceCreateFlags;
```

```
typedef VkFlags VkFormatFeatureFlags;
```

```
typedef VkFlags VkFramebufferCreateFlags;
```

```
typedef VkFlags VkImageAspectFlags;
```

```
typedef VkFlags VkImageCreateFlags;
```

```
typedef VkFlags VkImageUsageFlags;
```

```
typedef VkFlags VkImageViewCreateFlags;
```

```
typedef VkFlags VkIndirectCommandsLayoutUsageFlagsNVX;
```

```
typedef VkFlags VkInstanceCreateFlags;
```

```
typedef VkFlags VkMemoryAllocateFlagsKHX;
```

```
typedef VkFlags VkMemoryHeapFlags;
```

```
typedef VkFlags VkMemoryMapFlags;
```

```
typedef VkFlags VkMemoryPropertyFlags;
```

```
typedef VkFlags VkObjectEntryUsageFlagsNVX;
```

```
typedef VkFlags VkPeerMemoryFeatureFlagsKHX;
```

```
typedef VkFlags VkPipelineCacheCreateFlags;
```

```
typedef VkFlags VkPipelineColorBlendStateCreateFlags;
```

```
typedef VkFlags VkPipelineCreateFlags;
```

```
typedef VkFlags VkPipelineDepthStencilStateCreateFlags;
```

```
typedef VkFlags VkPipelineDynamicStateCreateFlags;
```

```
typedef VkFlags VkPipelineInputAssemblyStateCreateFlags;
```

```
typedef VkFlags VkPipelineLayoutCreateFlags;
```

```
typedef VkFlags VkPipelineMultisampleStateCreateFlags;
```

```
typedef VkFlags VkPipelineRasterizationStateCreateFlags;
```

```
typedef VkFlags VkPipelineShaderStageCreateFlags;
```

```
typedef VkFlags VkPipelineStageFlags;
```

```
typedef VkFlags VkPipelineTessellationStateCreateFlags;
```

```
typedef VkFlags VkPipelineVertexInputStateCreateFlags;
```

```
typedef VkFlags VkPipelineViewportStateCreateFlags;
```

```
typedef VkFlags VkPipelineViewportSwizzleStateCreateFlagsNV;
```

```
typedef VkFlags VkPipelineDiscardRectangleStateCreateFlagsEXT;
```

```
typedef VkFlags VkQueryControlFlags;
```

```
typedef VkFlags VkQueryPipelineStatisticFlags;
```

```
typedef VkFlags VkQueryPoolCreateFlags;
```

```
typedef VkFlags VkQueryResultFlags;
```

```
typedef VkFlags VkQueueFlags;
```

```
typedef VkFlags VkRenderPassCreateFlags;
```

```
typedef VkFlags VkSampleCountFlags;
```

```
typedef VkFlags VkSamplerCreateFlags;
```

```
typedef VkFlags VkSemaphoreCreateFlags;
```

```
typedef VkFlags VkShaderModuleCreateFlags;
```

```
typedef VkFlags VkShaderStageFlags;
```

```
typedef VkFlags VkSparseImageFormatFlags;
```

```
typedef VkFlags VkSparseMemoryBindFlags;
```

```
typedef VkFlags VkStencilFaceFlags;
```

```
typedef VkFlags VkSubpassDescriptionFlags;
```

```
typedef VkFlags VkSurfaceCounterFlagsEXT;
```

# Macro Definitions in vulkan.h

Vulkan is defined as a C API. The supplied vulkan.h header defines a small number of C preprocessor macros that are described below.

## Vulkan Version Number Macros

API Version Numbers are packed into integers. These macros manipulate version numbers in useful ways.

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) & 0xfff)
```

VK_API_VERSION_1_0 returns the API version number for Vulkan 1.0. The patch version number in this macro will always be zero. The supported patch version for a physical device **can** be queried with vkGetPhysicalDeviceProperties.

```
// Vulkan 1.0 version number
#define VK_API_VERSION_1_0 VK_MAKE_VERSION(1, 0, 0)
```

VK_API_VERSION is now commented out of vulkan.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)
```

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.

This macro **can** be used when constructing the VkApplicationInfo::apiVersion parameter passed to vkCreateInstance.

## Vulkan Header File Version Number

VK_HEADER_VERSION is the version number of the vulkan.h header. This value is currently kept synchronized with the release number of the Specification. However, it is not guaranteed to remain synchronized, since most Specification updates have no effect on vulkan.h.

```
// Version of this file
#define VK_HEADER_VERSION 50
```

## 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.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 the vulkan.h header 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. Behavior is undefined if the application defines a non-binary-compatible handle and **may** result in memory corruption or application termination. Binary compatibility is platform dependent so the application **must** be careful if it overrides the default `VK_DEFINE_NON_DISPATCHABLE_HANDLE` definition.

`VK_NULL_HANDLE` is a reserved value representing a non-valid object handle. It may be passed to and returned from Vulkan commands only when specifically allowed.

```
#define VK_NULL_HANDLE 0
```

# Platform-Specific Macro Definitions in vk_platform.h

Additional platform-specific macros and interfaces are defined using the included vk_platform.h file. 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.

## Platform-Specific Header Control

If the VK_NO_STDINT_H macro is defined by the application at compile time, extended integer types used by vulkan.h, such as `uint8_t`, **must** also be defined by the application. Otherwise, vulkan.h will not compile. If VK_NO_STDINT_H is not defined, the system <stdint.h> is used to define these types, or there is a fallback path when Microsoft Visual Studio version 2008 and earlier versions are detected at compile time.

## Window System-Specific Header Control

To use a Vulkan extension supporting a platform-specific window system, header files for that window systems **must** be included at compile time. The Vulkan header files cannot determine whether or not an external header is available at compile time, so applications wishing to use such an extension **must** #define a macro causing such headers to be included. If this is not done, the corresponding extension interfaces will not be defined and they will be unusable.

The extensions, **required** compile time symbols to enable them, window systems they correspond to, and external header files that are included when the macro is #defined are shown in the following table.

*Table 66. Window System Extensions and Required Compile Time Symbol Definitions*

| Extension Name | Required Compile Time Symbol | Window System Name | External Header Files Used |
|---|---|---|---|
| VK_KHR_android_surface | VK_USE_PLATFORM_ANDROID_KHR | Android Native | <android/native_window.h> |
| VK_KHR_mir_surface | VK_USE_PLATFORM_MIR_KHR | Mir | <mir_toolkit/client_types.h> |
| VK_KHR_wayland_surface | VK_USE_PLATFORM_WAYLAND_KHR | Wayland | <wayland-client.h> |
| VK_KHR_win32_surface | VK_USE_PLATFORM_WIN32_KHR | Microsoft Windows | <windows.h> |
| VK_KHR_xcb_surface | VK_USE_PLATFORM_XCB_KHR | X Window System Xcb library | <xcb/xcb.h> |
| VK_KHR_xlib_surface | VK_USE_PLATFORM_XLIB_KHR | X Window System Xlib library | <X11/Xlib.h> |
| VK_MVK_ios_surface | VK_USE_PLATFORM_IOS_MVK | iOS | None |
| VK_MVK_macos_surface | VK_USE_PLATFORM_MACOS_MVK | macOS | None |
| VK_NN_vi_surface | VK_USE_PLATFORM_VI_NN | VI | None |

# Appendix E: 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.

**Adjacent Vertex**

A vertex in an adjacency primitive topology that is not part of a given primitive, but is accessible in geometry shaders.

**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.

**API Order**

A set of ordering rules that govern how primitives in draw commands affect the framebuffer.

**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.

**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.

**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 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 Renderable Format**

A VkFormat where `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` is set in the `optimalTilingFeatures` or `linearTilingFeatures` field of VkFormatProperties::optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties, depending on the tiling used.

**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.

**Compatible Queues**

Queues within a queue family. Compatible queues have identical properties.

**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.

**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.

**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.

**Derivative Group**

A set of fragment 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 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 vkCmdDraw, and vkCmdDrawIndexed.

**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, vkCmdDrawIndirect, vkCmdDrawIndirectCountAMD,

vkCmdDrawIndexedIndirectCountAMD, 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 Size**

The size (in bytes) used to store one element of an uncompressed format or the size (in bytes) used to store one block of a block-compressed format.

**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.

**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 Input Attachment Interface**

A fragment shader entry point's variables with `UniformConstant` storage class and a decoration of `InputAttachmentIndex`, which receive values from input attachments.

**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.

**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 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. CPU 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.

**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,

vkCmdDrawIndexedIndirectCountAMD, and vkCmdDrawIndexedIndirect.

**Indirect Commands**

Drawing or dispatching commands that source some of their parameters from structures in buffer memory. Includes vkCmdDrawIndirect, vkCmdDrawIndexedIndirect, vkCmdDrawIndirectCountAMD, vkCmdDrawIndexedIndirectCountAMD, 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, vkCmdDrawIndirectCountAMD, vkCmdDrawIndexedIndirectCountAMD, and vkCmdDrawIndexedIndirect.

**Initial State (Command Buffer)**

A command buffer that has not begun recording commands. See also Recorded State and Executable State.

**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 Object**

High-level Vulkan objects, which are not logical devices, nor children of logical devices. For example, VkInstance and VkPhysicalDevice objects are instance-level objects.

**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 a VkBuffer, or a VkImage created with `VK_IMAGE_TILING_LINEAR`. A resource is *non-linear* if it is a VkImage created with `VK_IMAGE_TILING_OPTIMAL`.

**Local Workgroup**

A collection of compute shader invocations invoked by a single dispatch command, which share shared memory 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 hardware 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 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.

**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.

**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, 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.

**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 data element 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.

**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.

**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.

**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 `sname`: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 Topology**

State that controls how vertices are assembled into primitives, e.g. as lists of triangles, strips of lines, etc..

**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 that contains 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.

**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.

**Sampled Image**

A descriptor type that represents an image view, and supports filtered (sampled) and unfiltered read-only acccess in a shader.

**Sampler**

An object that contains 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.

**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.

**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**

The set of shader invocations exposed as running concurrently with the current shader invocation. 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, 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 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.

**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.

**Unnormalized**

A value that is interpreted according to its conventional interpretation, and is not normalized.

**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 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.

**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

# 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 F: Credits

Vulkan 1.0 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 contributions, are listed below. Some specific contributions made by individuals are listed together with their name.

- Adam Jackson, Red Hat
- Adam Śmigielski, Mobica
- Alex Bourd, Qualcomm Technologies, Inc.
- Alexander Galazin, ARM
- Allen Hux, Intel
- Alon Or-bach, Samsung Electronics (WSI technical sub-group chair)
- Andrew Cox, Samsung Electronics
- Andrew Garrard, Samsung Electronics (format wrangler)
- Andrew Poole, Samsung Electronics
- Andrew Rafter, Samsung Electronics
- Andrew Richards, Codeplay Software Ltd.
- Andrew Woloszyn, Google
- Antoine Labour, Google
- Aras Pranckevičius, Unity
- Ashwin Kolhe, NVIDIA
- Ben Bowman, Imagination Technologies
- Benj Lipchak
- Bill Hollings, The Brenwill Workshop
- Bill Licea-Kane, Qualcomm Technologies, Inc.
- Brent E. Insko, Intel
- Brian Ellis, Qualcomm Technologies, Inc.
- Cass Everitt, Oculus VR
- Cemil Azizoglu, Canonical
- Chad Versace, Intel
- Chang-Hyo Yu, Samsung Electronics
- Chia-I Wu, LunarG
- Chris Frascati, Qualcomm Technologies, Inc.
- Christophe Riccio, Unity

- Cody Northrop, LunarG
- Courtney Goeltzenleuchter, LunarG
- Damien Leone, NVIDIA
- Dan Baker, Oxide Games
- Dan Ginsburg, Valve
- Daniel Johnston, Intel
- Daniel Koch, NVIDIA (Shader Interfaces; Features, Limits, and Formats)
- Daniel Rakos, AMD
- David Airlie, Red Hat
- David Neto, Google
- David Mao, AMD
- David Yu, Pixar
- Dominik Witczak, AMD
- Frank (LingJun) Chen, Qualcomm Technologies, Inc.
- Fred Liao, Mediatek
- Gabe Dagani, Freescale
- Graeme Leese, Broadcom
- Graham Connor, Imagination Technologies
- Graham Sellers, AMD
- Hwanyong Lee, Kyungpook National University
- Ian Elliott, LunarG
- Ian Romanick, Intel
- James Jones, NVIDIA
- James Hughes, Oculus VR
- Jan Hermes, Continental Corporation
- Jan-Harald Fredriksen, ARM
- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA (extensive contributions, exhaustive review and rewrites for technical correctness)
- Jeff Juliano, NVIDIA
- Jeff Vigil, Qualcomm Technologies, Inc.
- Jens Owen, LunarG
- Jeremy Hayes, LunarG
- Jesse Barker, ARM
- Jesse Hall, Google

- Johannes van Waveren, Oculus VR
- John Kessenich, Google (SPIR-V and GLSL for Vulkan spec author)
- John McDonald, Valve
- Jon Ashburn, LunarG
- Jon Leech, Independent (XML toolchain, normative language, release wrangler)
- Jonas Gustavsson, Sony Mobile
- Jonathan Hamilton, Imagination Technologies
- Jungwoo Kim, Samsung Electronics
- Kenneth Benzie, Codeplay Software Ltd.
- Kerch Holt, NVIDIA (SPIR-V technical sub-group chair)
- Kristian Kristensen, Intel
- Krzysztof Iwanicki, Samsung Electronics
- Larry Seiler, Intel
- Lutz Latta, Lucasfilm
- Maria Rovatsou, Codeplay Software Ltd.
- Mark Callow
- Mark Lobodzinski, LunarG
- Mateusz Przybylski, Intel
- Mathias Heyer, NVIDIA
- Mathias Schott, NVIDIA
- Maxim Lukyanov, Samsung Electronics
- Maurice Ribble, Qualcomm Technologies, Inc.
- Michael Lentine, Google
- Michael Worcester, Imagination Technologies
- Michal Pietrasiuk, Intel
- Mika Isojarvi, Google
- Mike Stroyan, LunarG
- Minyoung Son, Samsung Electronics
- Mitch Singer, AMD
- Mythri Venugopal, Samsung Electronics
- Naveen Leekha, Google
- Neil Henning, Codeplay Software Ltd.
- Neil Trevett, NVIDIA
- Nick Penwarden, Epic Games
- Niklas Smedberg, Epic Games

- Norbert Nopper, Freescale
- Pat Brown, NVIDIA
- Patrick Doane, Blizzard Entertainment
- Peter Lohrmann, Valve
- Pierre Boudier, NVIDIA
- Pierre-Loup A. Griffais, Valve
- Piers Daniell, NVIDIA (dynamic state, copy commands, memory types)
- Piotr Bialecki, Intel
- Prabindh Sundareson, Samsung Electronics
- Pyry Haulos, Google (Vulkan conformance test subcommittee chair)
- Ray Smith, ARM
- Rob Stepinski, Transgaming
- Robert J. Simpson, Qualcomm Technologies, Inc.
- Rolando Caloca Olivares, Epic Games
- Roy Ju, Mediatek
- Rufus Hamede, Imagination Technologies
- Sean Ellis, ARM
- Sean Harmer, KDAB
- Shannon Woods, Google
- Slawomir Cygan, Intel
- Slawomir Grajewski, Intel
- Stefanus Du Toit, Google
- Steve Hill, Broadcom
- Steve Viggers, Core Avionics & Industrial Inc.
- Stuart Smith, Imagination Technologies
- Tim Foley, Intel
- Timo Suoranta, AMD
- Timothy Lottes, AMD
- Tobias Hector, Imagination Technologies (validity language and toolchain)
- Tobin Ehlis, LunarG
- Tom Olson, ARM (working group chair)
- Tomasz Kubale, Intel
- Tony Barbour, LunarG
- Wayne Lister, Imagination Technologies
- Yanjun Zhang, Vivante

- Zhenghong Wang, Mediatek

In addition to the Working Group, the Vulkan Advisory Panel members provided important real-world usage information and advice that helped guide design decisions.

Administrative support to the Working Group was provided by members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, Kathleen Mattson and Michelle Clark. Technical support was provided by James Riordon, webmaster of Khronos.org and OpenGL.org.