Vulkan API Reference Pages

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0.17	Fri Jun 17 15:45:17 PDT 2016	from git branch: 1.0 commit: 31018cf088c921ab90ec69dee033875807e8faec	

Contents

1 Drawing Commands

1.1 vkBeginCommandBuffer(3)

1.1.1 Name

vkBeginCommandBuffer - Start recording a command buffer

1.1.2 C Specification

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

1.1.3 Parameters

commandBuffer
A handle to the command buffer that is to be recorded.

pBeginInfo
A pointer to an instance of `VkCommandBufferBeginInfo` containing information about the command buffer.

1.1.4 Description

vkBeginCommandBuffer begins recording the command buffer whose handle is specified in *commandBuffer*. *pBeginInfo* is a pointer to an instance of the `VkCommandBufferBeginInfo` structure whose definition is:

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

The *sType* member of `VkCommandBufferBeginInfo` should be set to `VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO` and the *pNext* member of the structure is reserved for extensions and should be set to **NULL** if none are in use.

The *flags* member of *pBeginInfo* may be used to indicate the type of workload expected to be placed in the command buffer, which may allow implementations to optimize command buffer contents more appropriately. The available flags for use in this member are:

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

If the *flags* member contains `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`, then the command buffer may only be submitted to a queue for execution once, after which time it must be reset or destroyed. If this flag is not included, then it is legal to submit the command buffer many times.

If the *flags* member contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` then the command buffer must be a secondary command buffer (see `vkAllocateCommandBuffers` for more information) and is considered to be entirely contained inside a renderpass that is begun in the calling primary command buffer. In such a case, the *renderPass*, *subpass* and *framebuffer* members refer to the renderpass, subpass and framebuffer that will be active when the command buffer is referenced with a call to `vkCmdExecuteCommands`.

If VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT is not set, then the *renderPass*, *subpass* and *framebuffer* members are ignored. If the command buffer is a primary command buffer, then new renderpasses may be initiated by calls to vkCmdBeginRenderPass. If the command buffer is a secondary command buffer, then it may not contain commands that are legal only inside a renderpass, and may not be called from a primary command buffer while a renderpass is active.

If the *flags* member contains VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, then multiple submissions of the command buffer may be in flight simultaneously.

- A primary command buffer is considered to be in flight from the time it is submitted to a queue by a call to vkQueueSubmit until the time it is retired (signaling the fence passed to the call to vkQueueSubmit).
- A secondary command buffer is considered to be in flight from a reference to it it is made from a primary command buffer using a call to vkCmdExecuteCommands until that primary command buffer is retired.

If VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT is not set, then only one invocation of the command buffer may be in flight at any time.

It should be noted that for primary command buffers, VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT implies possible multiple submission of the command buffer, suggesting that VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT should be clear. Secondary command buffers may be referenced at most once from a primary command buffer (even the same primary command buffer) unless VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT is set.

The *renderPass* and *framebuffer* members must be **VK_NULL_HANDLE** for primary command buffers. For secondary command buffers, they must refer to the render pass and framebuffer that will be active when the secondary command buffer is called.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pBeginInfo* must be a pointer to a valid VkCommandBufferBeginInfo structure
- *commandBuffer* must not be in the recording state
- *commandBuffer* must not currently be pending execution
- 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

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

1.1.5 See Also

vkAllocateCommandBuffers, vkFreeCommandBuffers, vkEndCommandBuffer, vkResetCommandBuffer

1.2 vkBindBufferMemory(3)

1.2.1 Name

vkBindBufferMemory - Bind device memory to a buffer object

1.2.2 C Specification

```
VkResult vkBindBufferMemory(
    VkDevice                                    device,
    VkBuffer                                    buffer,
    VkDeviceMemory                              memory,
    VkDeviceSize                                memoryOffset);
```

1.2.3 Parameters

device
A handle to the device that owns the object to which memory will be bound.

buffer
A handle to the object to which to bind memory.

memory
A handle to the device memory object.

memoryOffset
The offset within the device memory object at which the binding should begin.

1.2.4 Description

vkBindBufferMemory binds a region of the device memory object specified by *memory* to the resource buffer specified by *buffer*. *buffer* must be the handle of a buffer resource.

memoryOffset specifies the offset within *memory*, in bytes, from which the binding will begin. The value of *memoryOffset* must satisfy the alignment requirments of the object specified in *buffer*. This value is returned in the *alignment* member of the VkMemoryRequirements retrieved by calling vkGetBufferMemoryRequirements with with *buffer* as specified.

vkBindBufferMemory should be used only for non-sparse resources. Memory is bound to sparse buffers by calling vkQueueBindSparse.

Valid Usage

- *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*
- Each of *device*, *buffer* and *memory* must have been created, allocated or retrieved from the same `VkPhysicalDevice`
- *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*
- The size of *buffer* must be less than or equal to the size of *memory* minus *memoryOffset*
- *memoryOffset* must be an integer multiple of the *alignment* member of the `VkMemoryRequirements` structure returned from a call to **vkGetBufferMemoryRequirements** with *buffer*

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`

1.2.5 See Also

vkQueueBindSparse

1.3 vkBindImageMemory(3)

1.3.1 Name

vkBindImageMemory - Bind device memory to an image object

1.3.2 C Specification

```
VkResult vkBindImageMemory(
    VkDevice                                    device,
    VkImage                                     image,
    VkDeviceMemory                              memory,
    VkDeviceSize                                memoryOffset);
```

1.3.3 Parameters

device
A handle to the device that owns the object to which memory will be bound.

image
A handle to the object to which to bind memory.

memory
A handle to the device memory object.

memoryOffset
The offset within the device memory object at which the binding should begin.

1.3.4 Description

vkBindImageMemory binds a region of the device memory object specified by *memory* to the resource image specified by *image*. *image* must be the handle of an image resource.

memoryOffset specifies the offset within *memory*, in bytes, from which the binding will begin. The value of *memoryOffset* must satisfy the alignment requirments of the image specified in *image*. This value is returned in the *alignment* member of the VkMemoryRequirements retrieved by calling vkGetImageMemoryRequirements with with *image* as specified.

vkBindImageMemory should be used only for non-sparse resources. Memory is bound to sparse imagesby calling vkQueueBindSparse.

Valid Usage

- *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*
- Each of *device*, *image* and *memory* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *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*

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

1.3.5 See Also

vkQueueBindSparse

1.4 vkCmdBeginQuery(3)

1.4.1 Name

vkCmdBeginQuery - Begin a query.

1.4.2 C Specification

```
void vkCmdBeginQuery(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    query,
    VkQueryControlFlags                         flags);
```

1.4.3 Parameters

commandBuffer
The command buffer upon which to execute the query.

queryPool
The query pool which contains the requested query.

entry
The index of the entry within *queryPool* at which the query resides.

flags
A set of flags controlling how the query should be executed (see VkQueryControlFlags).

1.4.4 Description

vkCmdBeginQuery begins the query located at the entry indicated by *entry* in the pool specified in *queryPool*. The *flags* parameter specifies how the query should be executed and must be one of the flags defined in VkQueryControlFlags.

Valid Usage

- *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
- Each of *commandBuffer* and *queryPool* must have been created, allocated or retrieved from the same VkDevice
- 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 created 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 created 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 created from must support compute operations

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS COMPUTE

1.4.5 Notes

Although **vkCmdBeginQuery** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.4.6 See Also

vkCmdEndQuery, vkCreateQueryPool, vkCmdResetQueryPool, vkCmdCopyQueryPoolResults, vkGetQueryPoolResults, vkDestroyQueryPool

1.5 vkCmdBeginRenderPass(3)

1.5.1 Name

vkCmdBeginRenderPass - Begin a new render pass.

1.5.2 C Specification

```
void vkCmdBeginRenderPass(
    VkCommandBuffer                             commandBuffer,
    const VkRenderPassBeginInfo*                pRenderPassBegin,
    VkSubpassContents                           contents);
```

1.5.3 Parameters

commandBuffer
The command buffer in which to begin the render pass.

pRenderPassBegin
A pointer to a structure describing how to begin the render pass.

contents
A description of how the commands for the first subpass of the render pass will be issued.

1.5.4 Description

vkCmdBeginRenderPass begins the first subpass of a new render pass in the command buffer specified by *commandBuffer*. Information about how to begin the render pass is given in an instance of the VkRenderPassBeginInfo structure, a pointer to which is specified in *pRenderPassBegin*. The definition of VkRenderPassBeginInfo is:

```
typedef struct VkRenderPassBeginInfo {
    VkStructureType        sType;
    const void*            pNext;
    VkRenderPass           renderPass;
    VkFramebuffer          framebuffer;
    VkRect2D               renderArea;
    uint32_t               clearValueCount;
    const VkClearValue*    pClearValues;
} VkRenderPassBeginInfo;
```

The *contents* parameter describes how the commands in the first subpass will be provided. If it is VK_SUBPASS_CONTENTS_INLINE, the contents of the subpass will be recorded inline in the primary command buffer, and calling a secondary command buffer within the subpass is an error. 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**.

vkCmdBeginRenderPass is only allowed in primary command buffers. A render pass must end in the same command buffer in which it was begun.

Valid Usage

- `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`
- 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 one of `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`, 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`.

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary	Outside	GRAPHICS

1.5.5 Notes

Although **vkCmdBeginRenderPass** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.5.6 See Also

vkCmdEndRenderPass, vkCreateRenderPass, vkDestroyRenderPass, vkCmdNextSubpass, vkCmdEndRenderPass

1.6 vkCmdBindDescriptorSets(3)

1.6.1 Name

vkCmdBindDescriptorSets - Binds descriptor sets to a command buffer.

1.6.2 C Specification

```
void vkCmdBindDescriptorSets(
    VkCommandBuffer                             commandBuffer,
    VkPipelineBindPoint                         pipelineBindPoint,
    VkPipelineLayout                            layout,
    uint32_t                                    firstSet,
    uint32_t                                    descriptorSetCount,
    const VkDescriptorSet*                      pDescriptorSets,
    uint32_t                                    dynamicOffsetCount,
    const uint32_t*                             pDynamicOffsets);
```

1.6.3 Parameters

commandBuffer
: The command buffer into which the command is to be placed.

pipelineBindPoint
: The pipeline bind point the descriptor sets should be bound to.

layout
: A handle to the layout used to create the descriptor sets.

firstSet
: The first descriptor set index of the pipeline bind point updated by the command.

descriptorSetCount
: The number of descriptor set index of the pipeline bind point updated by the command.

pDescriptorSets
: An array of *descriptorSetCount* number of descriptor set objects to bind.

dynamicOffsetCount
: The number of dynamic offsets to be applied to the descriptor sets.

pDynamicOffsets
: An array of *dynamicOffsetCount* number of offsets, each corresponding to a dynamic buffer descriptor in the specified descriptor sets.

1.6.4 Description

vkCmdBindDescriptorSets updates *descriptorSetCount* number of descriptor set bindings of the pipeline bind point specified by *pipelineBindPoint* starting from descriptor set index specified by *firstSet*. The parameter *pDescriptorSets* specifies an array of *descriptorSetCount* number of descriptor set objects to bind.

pDynamicOffsets provides *dynamicOffsetCount* number of offsets used for the dynamic buffer descriptors in the specified descriptor sets. Each offset corresponds to one dynamic buffer descriptor entry in the set index range. The order the offsets should be specified so that offsets corresponding to lower indexed sets appear before offsets corresponding to higher indexed sets, while offsets of the same set index should be specified so that offsets corresponding to lower indexed bindings appear before offsets corresponding to higher indexed bindings.

Valid Usage

- *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`
- Any given element of *pDescriptorSets* must have been created with a `VkDescriptorSetLayout` that matches (is the same as, or defined identically to) 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*
- *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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS COMPUTE

1.6.5 See Also

`vkAllocateDescriptorSets`, `vkFreeDescriptorSets`, `vkCreateDescriptorPool`, `vkCreateDescriptorSetLayout`

1.7 vkCmdBindIndexBuffer(3)

1.7.1 Name

vkCmdBindIndexBuffer - Bind an index buffer to a command buffer.

1.7.2 C Specification

```
void vkCmdBindIndexBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset,
    VkIndexType                                 indexType);
```

1.7.3 Parameters

commandBuffer
Specifies the command buffer to which to bind the index buffer.

buffer
The buffer object to bind.

offset
The offset from the start of the buffer object where index data begins.

indexType
The type of the index data stored in the buffer.

1.7.4 Description

vkCmdBindIndexBuffer binds the buffer object specified by *buffer*, starting at the byte offset specified in *offset* as an index buffer on the graphics pipeline bind point on *commandBuffer*. *indexType* specifies the type of the index data and must be one of VK_INDEX_TYPE_UINT16 or VK_INDEX_TYPE_UINT32, to indicate 16- or 32-bit unsigned data, respectively.

If *indexType* is VK_INDEX_TYPE_UINT16, then *offset* must be a multiple of two. If *indexType* is VK_INDEX_TYPE_UINT32, then offset must be a multiple of four.

Valid Usage

- *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
- Each of *commandBuffer* and *buffer* must have been created, allocated or retrieved from the same VkDevice
- *offset* must be less than the size of *buffer*
- The sum of *offset*, and the address of the range of VkDeviceMemory object that's 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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.7.5 Notes

Although **vkCmdBindIndexBuffer** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.7.6 See Also

vkCmdDrawIndexed

1.8 vkCmdBindPipeline(3)

1.8.1 Name

vkCmdBindPipeline - Bind a pipeline object to a command buffer.

1.8.2 C Specification

```
void vkCmdBindPipeline(
    VkCommandBuffer                             commandBuffer,
    VkPipelineBindPoint                         pipelineBindPoint,
    VkPipeline                                  pipeline);
```

1.8.3 Parameters

commandBuffer
The command buffer to which to bind the pipeline.

pipelineBindPoint
The pipeline bind point on the command buffer to which to bind the pipeline.

pipeline
The pipeline object to bind to *commandBuffer*.

1.8.4 Description

vkCmdBindPipeline binds the pipeline object specified in *pipeline* to the command buffer specified in *commandBuffer* at the bind point specified by *pipelineBindPoint*. The value of *pipelineBindPoint* must be supported by the command buffer, and be valid for the specified pipeline object. *pipelineBindPoint* may be one of VK_PIPELINE_BIND_POINT_COMPUTE or VK_PIPELINE_BIND_POINT_GRAPHICS, assuming the command buffer supports the corresponding bind point. All work subsequently issued in *commandBuffer* will use the pipeline bound to the corresponding pipeline bind point.

Valid Usage

- *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
- Each of *commandBuffer* and *pipeline* must have been created, allocated or retrieved from the same VkDevice
- 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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS COMPUTE

1.8.5 Notes

Although **vkCmdBindPipeline** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.8.6 See Also

vkCreateGraphicsPipelines, vkCreateComputePipelines

1.9 vkCmdBindVertexBuffers(3)

1.9.1 Name

vkCmdBindVertexBuffers - Bind vertex buffers to a command buffer

1.9.2 C Specification

```
void vkCmdBindVertexBuffers(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstBinding,
    uint32_t                                    bindingCount,
    const VkBuffer*                             pBuffers,
    const VkDeviceSize*                         pOffsets);
```

1.9.3 Parameters

commandBuffer
The first parameter.

startBinding
The index of the first vertex buffer binding to which to bind a vertex buffer.

bindingCount
The number of consecutive vertex buffer bindings to update.

pBuffers
A pointer to an array of `VkBuffer` handles representing the buffers to be bound.

pOffsets
A pointer to an array of `VkDeviceSize` values containing the offsets, in bytes, of each binding within its respective buffer.

1.9.4 Description

vkCmdBindVertexBuffers binds one or more vertex buffers to the command buffer specified by *commandBuffer*. The first binding to update is specified in *startBinding* and the number of bindings to update is specified in *bindingCount*.

pBuffers points to an array of *bindingCount* buffer object handles representing the buffers to bind. The same buffer may be referenced multiple times. *pOffsets* points to an array of *bindingCount* values containing the offsets, in bytes of the start of each binding within the current buffer.

Vertex data consumed by drawing commands such as `vkCmdDraw` or `vkCmdDrawIndexed` subsequently issued in *commandBuffer* is drawn from the buffers bound to that command buffer.

Valid Usage

- *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`
- Each of *commandBuffer* and the elements of *pBuffers* must have been created, allocated or retrieved from the same `VkDevice`
- *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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.9.5 See Also

`vkCmdBindIndexBuffer`, `vkCmdDraw`, `vkCmdDrawIndexed`, `vkCmdDrawIndirect`, `vkCmdDrawIndexedIndirect`

1.10 vkCmdBlitImage(3)

1.10.1 Name

vkCmdBlitImage - copies regions of an image, potentially performing format conversion, arbitrary scaling, and filtering (but does not allow MSAA resolve).

1.10.2 C Specification

```
void vkCmdBlitImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkImage                                     dstImage,
    VkImageLayout                               dstImageLayout,
    uint32_t                                    regionCount,
    const VkImageBlit*                          pRegions,
    VkFilter                                    filter);
```

1.10.3 Parameters

commandBuffer
The command buffer into which the command is to be placed.

srcImage
The image that is the source of the blit operation.

srcImageLayout
The layout of the source image at the time of the blit.

dstImage
The image into which image data is to be copied.

dstImageLayout
The layout of the destination image at the time of the blit.

regionCount
The number of regions to blit.

pRegions
An array of image regions to blit.

filter
Filtering operation to perform on the image while performing the blit.

1.10.4 Description

vkCmdBlitImage copies regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering (but does not allow MSAA resolve). The source and destination images are specified in *srcImage* and *dstImage*, respectively. The layout of the source and destination images must be provided in *srcImageLayout* and *dstImageLayout*, respectively. The *srcImageLayout* must be either VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL. The *dstImageLayout* must be either VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL.

The *srcImage* must support the VkFormatFeatureFlags bit VK_FORMAT_FEATURE_BLIT_SRC_BIT and the *dstImage* must support the VkFormatFeatureFlags bit VK_FORMAT_FEATURE_BLIT_DST_BIT. *srcImage* and *dstImage* may reference the same image but results are undefined if source and destination regions overlap. *srcImage* or *dstImage* may not refer to multi-sampled images. Use **vkCmdResolveImage** to resolve multi-sampled images.

pRegions is a pointer to an array of *regionCount* VkImageBlit structures, the definition of each is:

```
typedef struct VkImageBlit {
    VkImageSubresourceLayers    srcSubresource;
    VkOffset3D                  srcOffsets[2];
    VkImageSubresourceLayers    dstSubresource;
    VkOffset3D                  dstOffsets[2];
} VkImageBlit;
```

The *srcSubresource* and *dstSubresource* members of `VkImageBlit` specify the source and destination sub-resources, respectively. Each is a instance of the `VkImageSubresourceLayers` structure, the definition of which is:

```
typedef struct VkImageSubresourceLayers {
    VkImageAspectFlags    aspectMask;
    uint32_t              mipLevel;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceLayers;
```

The *srcOffset*[0] and *srcOffset*[1] members of `VkImageBlit` define the region of the source image to copy from, and the *dstOffset*[0] and *dstOffset*[1] members define the region of the destination image to copy to. The offset members are instances of the `VkOffset3D` structure. The definition of `VkOffset3D` is:

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

The size of the two regions need not match. If they are different, then the *filter* parameter determines the filtering mode used to expand or shrink the source region to fit the destination region. This is a member of the `VkFilter` enumeration, the definition of which is:

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

If the format of *srcImage* is an integer-based format then *filter* must be `VK_FILTER_NEAREST`.

Pixels are copied from the regions bound by srcOffset[0], srcOffset[1] to the region bound by destOffset[0], destOffset[1], scaling the result if the regions are different sizes.

`vkCmdBlitImage` does not perform any implicit barriers. Therefore, if any region in the array of *pRegions* a references updates from a prior region, then results are undefined.

This command may not be used within a renderpass.

Valid Usage

- *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*, *srcImage* and *dstImage* must have been created, allocated or retrieved from the same VkDevice
- 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_BLIT_SRC_BIT, which is indicated by VkFormatProperties::*linearTilingFeatures* (for linear 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
- *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 either of 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 linear 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
- *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 either of 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
- 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

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS

1.10.5 Notes

Although **`vkCmdBlitImage`** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.10.6 See Also

`vkCmdResolveImage`, `vkCmdCopyImage`

1.11 vkCmdClearAttachments(3)

1.11.1 Name

vkCmdClearAttachments - Clear regions within currently bound framebuffer attachments.

1.11.2 C Specification

```
void vkCmdClearAttachments(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    attachmentCount,
    const VkClearAttachment*                    pAttachments,
    uint32_t                                    rectCount,
    const VkClearRect*                          pRects);
```

1.11.3 Parameters

commandBuffer
The command buffer into which to insert the command.

attachmentCount
The number of attachments to clear.

pAttachments
A pointer to an array of structures describing the attachments to clear and the values to clear them to.

rectCount
The number of regions within the attachments to clear.

pRects
A pointer to an array of rectangles defining the regions to clear.

1.11.4 Description

vkCmdClearAttachments clears regions within the attachments associated with the current renderpass. *commandBuffer* is a handle to the command buffer into which to insert the command. A renderpass must be active on *commandBuffer*. *attachmentCount* specifies the number of attachments to clear and *pAttachments* is a pointer to an array of *attachmentCount* VkClearAttachment structures, each containing the aspect(s), attachment index and the clear value for each attachment. The definition of VkClearAttachment is:

```
typedef struct VkClearAttachment {
    VkImageAspectFlags    aspectMask;
    uint32_t              colorAttachment;
    VkClearValue          clearValue;
} VkClearAttachment;
```

aspectMask is a bitfield specifying the aspect or aspects to clear on the referenced attachment. It is constructed from a bitwise combination of the members of the VkImageAspectFlagBits enumeration, the definition of which is:

```
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 `colorAttachment` member of `VkClearAttachment` specifies the index of the color attachment within the current framebuffer. The `clearValue` member contains the value to which to clear the attachment. It is an instance of the `VkClearValue` union, the definition of which is:

```
typedef union VkClearValue {
    VkClearColorValue           color;
    VkClearDepthStencilValue    depthStencil;
} VkClearValue;
```

If the attachment and aspect referenced by `aspectMask` and `colorAttachment` is a color attachment, the values contained in the `color` field of `VkClearValue` is used to clear the attachment regions. If the attachment and aspect referenced by `aspectMask` and `colorAttachment` is a depth, stencil or depth-stencil attachment, then the `depthStencil` field of `VkClearValue` is used to clear the attachment.

The `rectCount` parameter to **`vkCmdClearAttachments`** specifies the number of regions of the attachments to clear. `pRects` is a pointer to an array of `rectCount` `VkClearRect` structures defining those regions. The definition of `VkClearRect` is:

```
typedef struct VkClearRect {
    VkRect2D    rect;
    uint32_t    baseArrayLayer;
    uint32_t    layerCount;
} VkClearRect;
```

The `rect` member of `VkClearRect` specifies the rectangle, measured in pixels, of the rectangle to clear. `baseArrayLayer` and `layerCount` specify the first layer and number of layers to clear and should be used to clear multiple layers in layered attachments.

Valid Usage

- `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`
- 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

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Inside	GRAPHICS

1.11.5 See Also

VkClearAttachment, vkCmdBeginRenderPass

1.12 vkCmdClearColorImage(3)

1.12.1 Name

vkCmdClearColorImage - Clear regions of a color image.

1.12.2 C Specification

```
void vkCmdClearColorImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     image,
    VkImageLayout                               imageLayout,
    const VkClearColorValue*                    pColor,
    uint32_t                                    rangeCount,
    const VkImageSubresourceRange*              pRanges);
```

1.12.3 Parameters

commandBuffer
: The command buffer into which the clear command is to be placed.

image
: The image that contains the regions to be cleared.

imageLayout
: The layout of the image at the time of the clear operation.

pColor
: A pointer to a structure containing the color with which to clear the image.

rangeCount
: The number of ranges to clear.

pRanges
: A pointer to an array of structures defining the regions to be cleared.

1.12.4 Description

vkCmdClearColorImage clears *rangeCount* regions of an image. The color with which to clear the image is specified an instance of the VkClearColorValue union pointed to by *pColor*. The definition of VkClearColorValue is:

```
typedef union VkClearColorValue {
    float       float32[4];
    int32_t     int32[4];
    uint32_t    uint32[4];
} VkClearColorValue;
```

The *float32*, *int32* and *uint32* members of *pColor* are arrays of four 32-bit floating point, signed integer or unsigned integer values, respectively. Which is used is determined from the format of the image specified in *image*.

The first element of the selected array is written to the first component of the target image, the second element to the second component, the third to the third and the fourth to the fourth, if those components exist. *pRanges* describes the regions to be cleared and points to an array of *rangeCount* VkImageSubresourceRange structures, the definition of which is:

```
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags    aspectMask;
    uint32_t              baseMipLevel;
    uint32_t              levelCount;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceRange;
```

Valid Usage

- *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
- Each of *commandBuffer* and *image* must have been created, allocated or retrieved from the same VkDevice
- *image* must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT usage flag
- *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 not have a compressed or depth/stencil format

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS COMPUTE

1.12.5 Notes

Although **vkCmdClearColorImage** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.12.6 See Also

vkCmdClearDepthStencilImage

1.13 vkCmdClearDepthStencilImage(3)

1.13.1 Name

vkCmdClearDepthStencilImage - Fill regions of a combined depth-stencil image.

1.13.2 C Specification

```
void vkCmdClearDepthStencilImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     image,
    VkImageLayout                               imageLayout,
    const VkClearDepthStencilValue*             pDepthStencil,
    uint32_t                                    rangeCount,
    const VkImageSubresourceRange*              pRanges);
```

1.13.3 Parameters

commandBuffer
The command buffer into which the command is to be placed.

image
The image containing the regions to be cleared.

imageLayout
The layout of the image to be cleared.

pDepthStencil
A pointer to a structure containing the values to clear the image with.

rangeCount
The number of image regions to clear.

pRanges
A pointer to an array of *rangeCount* regions to clear.

1.13.4 Description

vkCmdClearDepthStencilImage clears *rangeCount* regions of a combined depth-stencil image to the values specified in the structure whose address is given in *pDepthStencil*. This is a pointer to an instance of the VkClearDepthStencilValue structure, the definition of which is:

```
typedef struct VkClearDepthStencilValue {
    float       depth;
    uint32_t    stencil;
} VkClearDepthStencilValue;
```

The *depth* and *stencil* members contain the value to clear the depth and stencil aspects of the image to, respectively. *imageLayout* specifies the layout of the image being cleared. *pRanges* points to an array *rangeCount* regions of the image are cleared, each of which is described by an instance of the VkImageSubresourceRange structure, the definition of which is:

```
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags    aspectMask;
    uint32_t              baseMipLevel;
    uint32_t              levelCount;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceRange;
```

Valid Usage

- *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`
- Each of *commandBuffer* and *image* must have been created, allocated or retrieved from the same `VkDevice`
- *image* must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- *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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS

1.13.5 Notes

Although **`vkCmdClearDepthStencilImage`** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.13.6 See Also

`vkCmdClearColorImage`

1.14 vkCmdCopyBuffer(3)

1.14.1 Name

vkCmdCopyBuffer - Copy data between buffer regions.

1.14.2 C Specification

```
void vkCmdCopyBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    srcBuffer,
    VkBuffer                                    dstBuffer,
    uint32_t                                    regionCount,
    const VkBufferCopy*                         pRegions);
```

1.14.3 Parameters

commandBuffer
The command buffer into which the copy command is to be placed.

srcBuffer
The buffer containing the data to be copied.

dstBuffer
The buffer into which data will be copied.

regionCount
The number of regions of data to copy.

pRegions
An array of *regionCount* regions of data to be copied.

1.14.4 Description

vkCmdCopyBuffer copies regions of data from a source buffer to a destination buffer. *regionCount* regions are copied from *srcBuffer* to *dstBuffer*. Each region is represented by a member of the *pRegions* array, which is an array of the VkBufferCopy structure, whose definition is:

```
typedef struct VkBufferCopy {
    VkDeviceSize    srcOffset;
    VkDeviceSize    dstOffset;
    VkDeviceSize    size;
} VkBufferCopy;
```

If any two or more regions within *pRegions* overlap, the resulting data will be undefined. It is recommended, but not required, that the regions given in *pRegions* start on multiples of four bytes and have a length which is a multiple of four bytes.

Valid Usage

- *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*, *srcBuffer* and *dstBuffer* must have been created, allocated or retrieved from the same `VkDevice`
- 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
- *dstBuffer* must have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	TRANSFER GRAPHICS COMPUTE

1.14.5 Notes

Although **`vkCmdCopyBuffer`** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.14.6 See Also

`vkCmdCopyImage`, `vkCmdCopyBufferToImage`, `vkCmdCopyImageToBuffer`

1.15 vkCmdCopyBufferToImage(3)

1.15.1 Name

vkCmdCopyBufferToImage - Copy data from a buffer into an image.

1.15.2 C Specification

```
void vkCmdCopyBufferToImage(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    srcBuffer,
    VkImage                                     dstImage,
    VkImageLayout                               dstImageLayout,
    uint32_t                                    regionCount,
    const VkBufferImageCopy*                    pRegions);
```

1.15.3 Parameters

commandBuffer
The command buffer into which the copy command is to be placed.

srcBuffer
The buffer from which data is to be sourced.

dstImage
The image that is to be the destination for the copy.

dstImageLayout
The image layout of the destination image at the time of the copy operation.

regionCount
The number of image regions to update.

pRegions
An array of *regionCount* regions to update.

1.15.4 Description

vkCmdCopyBufferToImage copies *regionCount* regions of data from *srcBuffer* into *dstImage*. *pRegions* points to an array of VkBufferImageCopy structures which describe the regions to be copied. The definition of VkBufferImageCopy is:

```
typedef struct VkBufferImageCopy {
    VkDeviceSize                bufferOffset;
    uint32_t                    bufferRowLength;
    uint32_t                    bufferImageHeight;
    VkImageSubresourceLayers    imageSubresource;
    VkOffset3D                  imageOffset;
    VkExtent3D                  imageExtent;
} VkBufferImageCopy;
```

Valid Usage

- *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*, *srcBuffer* and *dstImage* must have been created, allocated or retrieved from the same `VkDevice`
- 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 have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- *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 either of `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	TRANSFER GRAPHICS COMPUTE

1.15.5 Notes

Although **`vkCmdCopyBufferToImage`** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.15.6 See Also

vkCmdCopyBuffer, vkCmdCopyImageToBuffer

1.16 vkCmdCopyImage(3)

1.16.1 Name

vkCmdCopyImage - Copy data between images.

1.16.2 C Specification

```
void vkCmdCopyImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkImage                                     dstImage,
    VkImageLayout                               dstImageLayout,
    uint32_t                                    regionCount,
    const VkImageCopy*                          pRegions);
```

1.16.3 Parameters

commandBuffer
: The command buffer into which the copy command is to be placed.

srcImage
: The image that is the source for the data.

srcImageLayout
: The layout of the source image at the time of the copy operation.

dstImage
: The image that is to be the destination for the copy.

dstImageLayout
: The layout of the destination image at the time of the copy operation.

regionCount
: The number of regions to copy.

pRegions
: An array of *regionCount* regions to copy.

1.16.4 Description

vkCmdCopyImage copies *regionCount* regions of image data between *srcImage* and *dstImage*. Each region is described by an element of the array pointed to by *pRegions*, which is an array of VkImageCopy, the definition of which is:

```
typedef struct VkImageCopy {
    VkImageSubresourceLayers    srcSubresource;
    VkOffset3D                  srcOffset;
    VkImageSubresourceLayers    dstSubresource;
    VkOffset3D                  dstOffset;
    VkExtent3D                  extent;
} VkImageCopy;
```

Valid Usage

- *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*, *srcImage* and *dstImage* must have been created, allocated or retrieved from the same VkDevice
- 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 have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT usage flag
- *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 either of VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- *dstImage* must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT usage flag
- *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 either of 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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	TRANSFER GRAPHICS COMPUTE

1.16.5 Notes

Although **vkCmdCopyBufferToImage** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.16.6 See Also

vkCmdCopyBuffer, vkCmdCopyImageToBuffer, vkCmdCopyBufferToImage

1.17 vkCmdCopyImageToBuffer(3)

1.17.1 Name

vkCmdCopyImageToBuffer - Copy image data into a buffer.

1.17.2 C Specification

```
void vkCmdCopyImageToBuffer(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkBuffer                                    dstBuffer,
    uint32_t                                    regionCount,
    const VkBufferImageCopy*                    pRegions);
```

1.17.3 Parameters

commandBuffer
The command buffer into which the copy command is to be placed.

srcImage
The image that is the source for the data.

srcImageLayout
The layout of the source image at the time of the copy operation.

dstBuffer
The buffer that is to receive the copied data.

regionCount
The number of regions to copy.

pRegions
An array of *regionCount* regions to copy.

1.17.4 Description

vkCmdCopyImageToBuffer copies image data into a buffer object. *srcImage* specifies the image that is to be the source of the data. *dstBuffer* is the buffer into which the data is to be copied. *pRegions* points to an array of *regionCount* VkBufferImageCopy structures, the definition of which is:

```
typedef struct VkBufferImageCopy {
    VkDeviceSize                bufferOffset;
    uint32_t                    bufferRowLength;
    uint32_t                    bufferImageHeight;
    VkImageSubresourceLayers    imageSubresource;
    VkOffset3D                  imageOffset;
    VkExtent3D                  imageExtent;
} VkBufferImageCopy;
```

Valid Usage

- *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*, *srcImage* and *dstBuffer* must have been created, allocated or retrieved from the same `VkDevice`
- 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 have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag
- *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 either of `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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	TRANSFER GRAPHICS COMPUTE

1.17.5 Notes

Although **`vkCmdCopyImageToBuffer`** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.17.6 See Also

vkCmdCopyBufferToImage, vkCmdCopyImage

1.18 vkCmdCopyQueryPoolResults(3)

1.18.1 Name

vkCmdCopyQueryPoolResults - Copy the results of queries in a query pool to a buffer object.

1.18.2 C Specification

```
void vkCmdCopyQueryPoolResults(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount,
    VkBuffer                                    dstBuffer,
    VkDeviceSize                                dstOffset,
    VkDeviceSize                                stride,
    VkQueryResultFlags                          flags);
```

1.18.3 Parameters

commandBuffer
The command buffer into which the command is to be placed.

queryPool
The query pool whose results should be copied to the buffer object.

startQuery
The index of the first query in the query pool whose results should be copied to the buffer object.

queryCount
The number of queries in the query pool whose results should be copied to the buffer object.

dstBuffer
The buffer object the results should be written to.

dstOffset
The offset within the buffer object the results should be written to.

stride
The stride between subsequent query result writes.

flags
The flags controlling the behavior of the query result copy command (see `VkQueryResultFlags`).

1.18.4 Description

`vkCmdCopyQueryPoolResults` copies the results of *`queryCount`* number of queries in the query pool specified by *`queryPool`* starting from index *`startQuery`*. The results are written to the buffer object specified by *`dstBuffer`* starting from *`dstOffset`* with each subsequent query's result being written *`stride`* number of bytes after the previous one. The semantics of when and what values written to the destination buffer are defined by the type of the queries in the query pool, the query control flags passed to `vkCmdBeginQuery`, and the query result control flags specified by *`flags`*.

Valid Usage

- `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`, `queryPool` and `dstBuffer` must have been created, allocated or retrieved from the same `VkDevice`
- `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 the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TIMESTAMP`, `flags` must not contain `VK_QUERY_RESULT_PARTIAL_BIT`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS COMPUTE

1.18.5 Notes

Although **`vkCmdCopyQueryPoolResults`** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.18.6 See Also

`vkGetQueryPoolResults`, `vkCmdBeginQuery`, `vkCmdEndQuery`, `vkCmdResetQueryPool`, `vkDestroyQueryPool`, `vkCreateQueryPool`

1.19 vkCmdDispatch(3)

1.19.1 Name

vkCmdDispatch - Dispatch compute work items.

1.19.2 C Specification

```
void vkCmdDispatch(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    x,
    uint32_t                                    y,
    uint32_t                                    z);
```

1.19.3 Parameters

commandBuffer
Command buffer upon which to execute the command.

x
Number of workgroups to dispatch in the X dimension.

y
Number of workgroups to dispatch in the Y dimension.

z
Number of workgroups to dispatch in the Z dimension.

1.19.4 Description

vkCmdDispatch dispatches a *x* by *y* by *z* group of compute workgroups. Two- and one-dimensional work groups can be dispatched by setting the *z*, or *y* and *z* parameters to 1, respectively. The size of each workgroup is determined by the pipeline bound to the VK_PIPELINE_BIND_POINT_COMPUTE bind point on the command buffer specified by *commandBuffer*.

Valid Usage

- `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
- `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]
- 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 [?]
- 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 [?]
- 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`**

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	COMPUTE

1.19.5 Notes

Although **vkCmdDispatch** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.19.6 See Also

vkCmdDispatchIndirect

1.20 vkCmdDispatchIndirect(3)

1.20.1 Name

vkCmdDispatchIndirect - Dispatch compute work items using indirect parameters.

1.20.2 C Specification

```
void vkCmdDispatchIndirect(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset);
```

1.20.3 Parameters

commandBuffer
Command buffer upon which to execute the command.

buffer
The buffer object containing the parameters to dispatch.

offset
The offset within *buffer* at which the parameters are located.

1.20.4 Description

vkCmdDispatchIndirect dispatches a group of *x* by *y* by *z* compute workgroups where the values of *x*, *y*, and *z* are taken from *offset* bytes into the buffer object specified by *buffer*. At this location in the buffer, there is assumed to be an instance of the VkDispatchIndirectCommand structure, whose definition is:

```
typedef struct VkDispatchIndirectCommand {
    uint32_t    x;
    uint32_t    y;
    uint32_t    z;
} VkDispatchIndirectCommand;
```

offset must be a multiple of four. If any of the *x*, *y* or *z* members of VkDispatchIndirectCommand are zero, then no work is initiated. Two- and one-dimensional work may be initiated by setting *z* or *y* and *z* to 1, respectively.

Valid Usage

- `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
- Each of `commandBuffer` and `buffer` must have been created, allocated or retrieved from the same `VkDevice`
- 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 [?]
- 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 [?]
- 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`**

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	COMPUTE

1.20.5 Notes

Although **vkCmdDispatchIndirect** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.20.6 See Also

vkCmdDispatch, vkCmdDrawIndirect, vkCmdDrawIndexedIndirect

1.21 vkCmdDraw(3)

1.21.1 Name

vkCmdDraw - Draw primitives.

1.21.2 C Specification

```
void vkCmdDraw(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    vertexCount,
    uint32_t                                    instanceCount,
    uint32_t                                    firstVertex,
    uint32_t                                    firstInstance);
```

1.21.3 Parameters

commandBuffer
: The command buffer into which the drawing command is to be placed.

firstVertex
: The first vertex to be passed to the graphics pipeline.

vertexCount
: The number of vertices passed to the graphics pipeline.

firstInstance
: The first instance of data to be passed to the graphics pipeline.

instanceCount
: The number of instances to be passed to the graphics pipeline.

1.21.4 Description

vkCmdDraw invokes a draw in the bound graphics pipeline. *instanceCount* instances of *vertexCount* vertices are produced. The vertex index presented to the pipeline is automatically generated, starting from *firstVertex* and counting forwards. For each instance, the instance index is generated automaticall, starting from *firstInstance* and counting forwards. If *vertexCount* or *vertexCount* is zero, then no vertices are generated.

Valid Usage

- *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
- 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 [?]
- 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 [?]
- 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 [?]
- 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**

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Inside	GRAPHICS

1.21.5 Notes

Although **vkCmdDraw** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.21.6 See Also

vkCmdDrawIndexed

1.22 vkCmdDrawIndexed(3)

1.22.1 Name

vkCmdDrawIndexed - Issue an indexed draw into a command buffer.

1.22.2 C Specification

```
void vkCmdDrawIndexed(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    indexCount,
    uint32_t                                    instanceCount,
    uint32_t                                    firstIndex,
    int32_t                                     vertexOffset,
    uint32_t                                    firstInstance);
```

1.22.3 Parameters

commandBuffer
Specifies the command buffer into which to insert the draw command.

firstIndex
Specifies the first element from the index buffer to be consumed by the command.

indexCount
Specifies the number of elements from the index buffer to be consumed by the command.

vertexOffset
Specifies a constant offset to be added to the value retrieved from the index buffer.

firstInstance
Specifies the starting value of the internally generated instance count.

instanceCount
Specifies the number of instances of the geometry to consume.

1.22.4 Description

vkCmdDrawIndexed issues an indexed draw into a command bufer. The command consumes *indexCount* elements from the bound index buffer, starting from *firstIndex*, and inserts them into graphics pipeline. Before insertion to the pipeline, *vertexOffset* is added to each index value. *instanceCount* instances of the index buffer range are inserted into the pipeline. The first shader in the pipeline is presented with the instance index, which begins at *firstInstance*.

Valid Usage

- *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
- 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 [?]
- 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 [?]
- 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 [?]
- 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**

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Inside	GRAPHICS

1.22.5 Notes

Although **vkCmdDrawIndexed** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.22.6 See Also

vkCmdDraw, vkCmdDrawIndirect, vkCmdDrawIndexedIndirect

1.23 vkCmdDrawIndexedIndirect(3)

1.23.1 Name

vkCmdDrawIndexedIndirect - Perform an indexed indirect draw.

1.23.2 C Specification

```
void vkCmdDrawIndexedIndirect(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset,
    uint32_t                                    drawCount,
    uint32_t                                    stride);
```

1.23.3 Parameters

commandBuffer
The command buffer upon which to execute the command.

buffer
The buffer from which to source the indirect draw parameters.

offset
The offset within the buffer where the draw parameters are located.

drawCount
The number of draws to issue.

stride
The stride between each structure member.

1.23.4 Description

vkCmdDrawIndexedIndirect issues an indirect indexed draw list containing *drawCount* draws into the command buffer specified in *commandBuffer*. *buffer* is the buffer containing the drawing parameters, which begin at *offset* bytes into the buffer. Each command is an instance of a VkDrawIndexedIndirectCommand structure, separated by *stride* bytes in memory. If *stride* is zero, then the array is assumed to be tightly packed. The definition of VkDrawIndexedIndirectCommand is as follows.

```
typedef struct VkDrawIndexedIndirectCommand {
    uint32_t    indexCount;
    uint32_t    instanceCount;
    uint32_t    firstIndex;
    int32_t     vertexOffset;
    uint32_t    firstInstance;
} VkDrawIndexedIndirectCommand;
```

The members of VkDrawIndexedIndirectCommand are interpreted in the same fashion as the similarly named parameters of vkCmdDrawIndexed. *offset* and *stride* should be multiples of four.

Valid Usage

- *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
- Each of *commandBuffer* and *buffer* must have been created, allocated or retrieved from the same VkDevice
- *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**
- 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 [?]
- 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 [?]
- 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* x (*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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Inside	GRAPHICS

1.23.5 Notes

Although **vkCmdDrawIndexedIndirect** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.23.6 See Also

vkCmdDrawIndirect, vkCmdDrawIndexed, vkCmdDraw, vkCmdDispatchIndirect

1.24 vkCmdDrawIndirect(3)

1.24.1 Name

vkCmdDrawIndirect - Issue an indirect draw into a command buffer.

1.24.2 C Specification

```
void vkCmdDrawIndirect(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset,
    uint32_t                                    drawCount,
    uint32_t                                    stride);
```

1.24.3 Parameters

commandBuffer
Specifies the command buffer into which to insert the draw command.

buffer
Specifies a handle of a buffer object containing parameters forming individual draw commands.

offset
Specifies offset, in bytes, within the buffer object represented by *buffer* at which the drawing command parameters begin.

drawCount
Specifies the number of indirect draws to consume from the specified memory object.

stride
Specifies the distance, in bytes, between the start of each indirect draw in the memory object. This parameter may be zero to indicate that the array of indirect draw commands is tightly packed.

1.24.4 Description

vkCmdDrawIndirect issues an indirect draw into a command bufer. Each indirect command consumes *drawCount* structures, stored at *offset* bytes into the buffer object whose handle is specified in *buffer*. The beginning of each structure is *stride* bytes from the previous. The data structures have the a layout in memory which may be represented by the VkDrawIndirectCommand structure, the definition of which is:

```
typedef struct VkDrawIndirectCommand {
    uint32_t    vertexCount;
    uint32_t    instanceCount;
    uint32_t    firstVertex;
    uint32_t    firstInstance;
} VkDrawIndirectCommand;
```

If *stride* is zero, the array of VkDrawIndirectCommand structures is assumed to be tightly packed.

Valid Usage

- *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
- Each of *commandBuffer* and *buffer* must have been created, allocated or retrieved from the same VkDevice
- *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**
- 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 [?]
- 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 [?]
- 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* x (*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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Inside	GRAPHICS

1.24.5 Notes

Although **vkCmdDrawIndirect** does not generate errors or return a value, enabled validation layers may detect possible error conditions or potentially undefined behavior and report this via some other means.

1.24.6 See Also

vkCmdDraw, vkCmdDrawIndexed, vkCmdDrawIndexedIndirect

1.25 vkCmdEndQuery(3)

1.25.1 Name

vkCmdEndQuery - Ends a query.

1.25.2 C Specification

```
void vkCmdEndQuery(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    query);
```

1.25.3 Parameters

commandBuffer
The command buffer upon which to execute the command.

queryPool
The pool in which the query to be stopped resides.

entry
The entry within *queryPool* at which the query to be stopped resides.

1.25.4 Description

vkCmdEndQuery ends the query at the entry specified by *entry* in the query pool specified by *queryPool*. The command is executed in the command buffer specified by *commandBuffer*. The query referenced by *queryPool* and *entry* should be an active query for which **vkCmdBeginQuery** has been called in the past.

Valid Usage

- *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
- Each of *commandBuffer* and *queryPool* must have been created, allocated or retrieved from the same VkDevice
- The query identified by *queryPool* and *query* must currently be active
- *query* must be less than the number of queries in *queryPool*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS COMPUTE

1.25.5 See Also

vkCmdBeginQuery, vkCmdResetQueryPool, vkCreateQueryPool, vkDestroyQueryPool, vkGetQueryPoolResults, vkCmdCopyQueryPoolResults

1.26 vkCmdEndRenderPass(3)

1.26.1 Name

vkCmdEndRenderPass - End the current render pass.

1.26.2 C Specification

```
void vkCmdEndRenderPass(
    VkCommandBuffer                             commandBuffer);
```

1.26.3 Parameters

commandBuffer
A handle to the command buffer in which the render pass is to be ended.

1.26.4 Description

vkCmdEndRenderPass ends the current render pass in the command buffer specified by *commandBuffer*. A render pass must begin and end in the same command buffer.

vkCmdEndRenderPass is only allowed in primary command buffers.

When **vkCmdEndRenderPass** executes, the store op for all attachments in the render pass is performed, and the attachment images are transitioned to their final layout.

Valid Usage

- *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
- The current subpass index must be equal to the number of subpasses in the render pass minus one

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary	Inside	GRAPHICS

1.26.5 See Also

vkCmdBeginRenderPass, vkCreateRenderPass

1.27 vkCmdExecuteCommands(3)

1.27.1 Name

vkCmdExecuteCommands - Execute a secondary command buffer from a primary command buffer.

1.27.2 C Specification

```
void vkCmdExecuteCommands(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    commandBufferCount,
    const VkCommandBuffer*                      pCommandBuffers);
```

1.27.3 Parameters

commandBuffer
The primary command buffer from which to call the secondary command buffers.

commandBuffersCount
Length of the pCommandBuffers array.

pCommandBuffers
An array of secondary command buffer handles.

1.27.4 Description

vkCmdExecuteCommands executes the contents of the secondary command buffers, in the order they appear in the *pCommandBuffers* array.

If any of the secondary command buffers contains commands that may only be executed inside a renderpass, then they may only be executed between calls to vkCmdBeginRenderPass and vkCmdEndRenderPass and the active renderpass must have a VkSubpassContents property of VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS. The *commandBuffer* argument must be a primary command buffer. The renderpass and framebuffer provided when beginning the secondary command buffer must match the VkRenderPass and VkFramebuffer provided to **vkCmdBeginRenderPass**.

If a secondary command buffer was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT usage, then only a single call to the secondary command buffer may exist in any primary command buffer at one time. If this usage bit is clear, then the secondary command buffer may be called multiple times from the same or multiple primary command buffers.

A secondary command buffer must be finished recording, via **vkEndCommandBuffer**, before it can be referenced in a call to **vkCmdExecuteCommands**. It must not be reset or destroyed before primary command buffers referencing it have completed executing.

A secondary command buffer can safely be passed to multiple **vkCmdExecuteCommands** (affecting different primary command buffers) simultaneously, only if it was recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT flag.

Valid Usage

- `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`
- Each of `commandBuffer` and the elements of `pCommandBuffers` must have been created, allocated or retrieved from the same `VkDevice`
- `commandBuffer` must have been created with a `level` of `VK_COMMAND_BUFFER_LEVEL_PRIMARY`
- Any given element of `pCommandBuffers` must have been created with a `level` of `VK_COMMAND_BUFFER_LEVEL_SECONDARY`
- Any given element of `pCommandBuffers` must not be already pending execution in `commandBuffer`, or appear twice in `pCommandBuffers`, unless it was created with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag
- Any given element of `pCommandBuffers` must not be already pending execution in any other `VkCommandBuffer`, unless it was created with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag
- Any given element of `pCommandBuffers` must be in the executable state
- 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, any given element of `pCommandBuffers` must have been recorded with a render pass that is compatible with the current render pass - see [?]
- 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 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`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary	Both	TRANSFER GRAPHICS COMPUTE

1.27.5 See Also

`vkAllocateCommandBuffers`, `vkFreeCommandBuffers`, `vkCmdBeginRenderPass`, `vkCmdEndRenderPass`

1.28 vkCmdFillBuffer(3)

1.28.1 Name

vkCmdFillBuffer - Fill a region of a buffer with a fixed value.

1.28.2 C Specification

```
void vkCmdFillBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    dstBuffer,
    VkDeviceSize                                dstOffset,
    VkDeviceSize                                size,
    uint32_t                                    data);
```

1.28.3 Parameters

commandBuffer
The command buffer upon which to execute the command.

dstBuffer
The destination buffer.

dstOffset
The offset in the buffer at which to begin filling.

size
The size of the region to be filled, in bytes.

data
The data with which to fill the buffer region.

1.28.4 Description

vkCmdFillBuffer fills a region of a buffer object with the fixed, 32-bit pattern specified in *data*. The command is executed in *commandBuffer*. *dstBuffer* specifies the destination buffer object, *dstOffset* specifies the offset within the buffer at which to begin filling and *size* specifies the size of the region to be filled, in bytes. *dstOffset* and *size* must be multiples of four bytes.

Valid Usage

- *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 graphics or compute operations
- This command must only be called outside of a render pass instance
- Each of *commandBuffer* and *dstBuffer* must have been created, allocated or retrieved from the same VkDevice
- *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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS COMPUTE

1.28.5 See Also

vkCreateBuffer, vkCreateBufferView

1.29 vkCmdNextSubpass(3)

1.29.1 Name

vkCmdNextSubpass - Transition to the next subpass of a render pass.

1.29.2 C Specification

```
void vkCmdNextSubpass(
    VkCommandBuffer                             commandBuffer,
    VkSubpassContents                           contents);
```

1.29.3 Parameters

commandBuffer
: The command buffer in which to switch to the next subpass.

contents
: A description of how the commands for the next subpass will be issued.

1.29.4 Description

vkCmdNextSubpass finalizes the previous subpass of the current render pass and prepares for the next subpass. It may only be called in a primary command buffer when a render pass is active. For a render pass with N subpasses, **vkCmdNextSubpass** must be used exactly N-1 times between **vkCmdBeginRenderPass** and **vkCmdEndRenderPass** to transition through all of the subpasses.

The *contents* parameter describes how the commands in the next subpass will be provided. If it is VK_SUBPASS_CONTENTS_INLINE, the contents of the subpass will be recorded inline in the primary command buffer, and calling a secondary command buffer within the subpass is an error. 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**.

Transitioning between subpasses performs any multisample resolve operations in the pass being ended, and transitions attachment images from their current layout to the layout required by the next subpass.

Valid Usage

- *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
- The current subpass index must be less than the number of subpasses in the render pass minus one

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary	Inside	GRAPHICS

1.29.5 See Also

vkCmdEndRenderPass, vkCreateRenderPass

1.30 vkCmdPipelineBarrier(3)

1.30.1 Name

vkCmdPipelineBarrier - Insert a set of execution and memory barriers.

1.30.2 C Specification

```
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);
```

1.30.3 Parameters

commandBuffer
The command buffer in which to wait.

srcStageMask
Specifies which pipeline stages must complete executing prior commands (see `VkPipelineStageFlags` for more detail).

dstStageMask
Specifies which pipeline stages do not begin executing subsequent commands until the barrier completes (see `VkPipelineStageFlags` for more detail).

byRegion
Indicates whether the barrier has screen-space locality (described below).

memoryBarrierCount
Number of memory barriers to insert after waiting for the pipe events.

ppMemoryBarriers
Array of pointers to memory barrier structures specifying the parameters of the memory barriers to insert as part of the pipeline barrier. Each element of the array may point to a `VkMemoryBarrier`, `VkBufferMemoryBarrier`, or `VkImageMemoryBarrier` structure.

1.30.4 Description

`vkCmdPipelineBarrier` inserts a set of execution and memory barriers into the command buffer specified by *`commandBuffer`*. The number of barriers to insert is specified in *`memoryBarrierCount`* and the description of those barriers is specified in a number instances of the `VkMemoryBarrier`, `VkBufferMemoryBarrier` or `VkImageMemoryBarrier` structures. The definitions of these structures are:

```
typedef struct VkMemoryBarrier {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
} VkMemoryBarrier;
```

```
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;
```

```
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;
```

The *ppMemoryBarriers* parameter points to an array of pointers to these structures. Each element of *ppMemoryBarriers* may point to a different type of structure. The type of each structure is identified by its *sType* member member. This should be set to VK_STRUCTURE_TYPE_MEMORY_BARRIER, VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER or VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER for VkMemoryBarrier, VkBufferMemoryBarrier and VkImageMemoryBarrier, respectively.

Writes as described by *outputMask* that were written by pipeline stages in *srcStageMask* prior to the barrier are made visible to reads as described by *inputMask* in pipeline stages in *dstStageMask* subsequent to the barrier. If *byRegion* is true, then the writes are made visible only to work in the same (implementation-dependent) screen-space region. This effectively reqiures that the subsequent work only reads data written by the same fragment location in the previous work. *byRegion* should only be set to true when the *srcStageMask* and *dstStageMask* only include screen-space work (fragment shader, early and late fragment tests, and/or attachment outputs).

In case of global memory barriers inserted by passing an VkMemoryBarrier structure to the command prior writes in the requested pipeline stages to any memory location corresponding to the set of memory output coherency flags specified in the *outputMask* member of the structure are made coherent with subsequent reads in the requested pipeline stages of any memory location corresponding to the set of memory input coherency flags specified in the *inputMask* member of the structure.

In case of buffer memory barriers inserted by passing an VkBufferMemoryBarrier structure to the command prior writes in the requested pipeline stages to the specified sub-range of the buffer corresponding to the set of memory output coherency flags specified in the *outputMask* member of the structure are made coherent with subsequent reads in the requested pipeline stages of the specified sub-range of the buffer corresponding to the set of memory input coherency flags specified in the *inputMask* member of the structure.

In case of image memory barriers inserted by passing an VkImageMemoryBarrier structure to the command prior writes in the requested pipeline stages to the specified sub-range of the image corresponding to the set of memory output coherency flags specified in the *outputMask* member of the structure are made coherent with subsequent reads in the requested pipeline stages of the specified sub-range of the image corresponding to the set of memory input coherency flags specified in the *inputMask* member of the structure. Additionally, if the *oldLayout* and *newLayout* members of the structure don't match a layout transition is performed on the specified sub-range of the image as part of the memory barrier.

In case of buffer and image memory barriers the *srcQueueFamilyIndex* and *dstQueueFamilyIndex* members of the corresponding memory barrier structures can specify the parameters of a transfer of ownership between two distinct families of queues of a shared buffer or image object created with the VK_SHARING_MODE_EXCLUSIVE sharing mode. In case of regular resource transitions both *srcQueueFamilyIndex* and *dstQueueFamilyIndex* should be set to VK_QUEUE_FAMILY_IGN

ORED to indicate no transfer of ownership between queue families. In case of resource transitions involving ownership transfer of shared buffers or images one of these two members have to match the queue family index the command buffer specified by *commandBuffer* was created for, while the other should specify the queue family index the ownership transfer is released to or acquired from. Ownership transferring resource transitions have to be performed both on a queue from the source queue family and on a queue from the destination queue family (see VkSharingMode for more detail).

If the *inputMask* member is zero in any of the memory barrier structures then prior writes will only be coherent with any type of subsequent read after a future resource transition command specifies a non-empty set of memory input coherency control flags. This allows flushing device output caches unconditionally.

If the *outputMask* member is zero in any of the memory barrier structures then subsequent reads will only be coherent with any type of prior write if an earlier resource transition command specified a non-empty set of memory output coherency control flags. This allows invalidating device input caches unconditionally.

Valid Usage

- *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
- 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 declare at least one self-dependency from the current subpass to itself - see Subpass Self-dependency

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	TRANSFER GRAPHICS COMPUTE

1.30.5 See Also

vkCmdWaitEvents, VkMemoryBarrier, VkBufferMemoryBarrier, VkImageMemoryBarrier, VkPipelineStageFlags

1.31 vkCmdPushConstants(3)

1.31.1 Name

vkCmdPushConstants - Update the values of push constants.

1.31.2 C Specification

```
void vkCmdPushConstants(
    VkCommandBuffer                             commandBuffer,
    VkPipelineLayout                            layout,
    VkShaderStageFlags                          stageFlags,
    uint32_t                                    offset,
    uint32_t                                    size,
    const void*                                 pValues);
```

1.31.3 Parameters

commandBuffer
A handle to the command buffer into which to insert the command.

layout
A handle to the pipeline layout describing the layout of the push constants.

stageFlags
A bitmask specifying the pipeline stages for which to update push constants.

offset
The offset of the first push constant to update in the layout.

size
The size of the push constants to update.

pValues
A pointer to a region of memory containing the new values for the push constants.

1.31.4 Description

vkCmdPushConstants updates the values of push constants for the command buffer specified by *commandBuffer*. Push constants become visible to the next drawing or dispatch command appended to *commandBuffer*. *layout* specifies a handle to a pipeline layout object containing the layout information for the push constants. *stageFlags* specifies the pipeline stages for which the push constant update is to be applied. This parameter is a bitwise combination of members of the VkShaderStageFlagBits enumeration and must match the shader stages used in the pipeline layout for the range specified by *offset* and *size*. The definition of VkShaderStageFlagBits is:

```
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;
```

`offset` and `size` specify the offset of the start of the region to be updated and its size, respectively. Both are in units of bytes.

`pValues` is a pointer to a region of `size` bytes of memory containing the new values for the specified push constants.

Valid Usage

- `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`
- Each of `commandBuffer` and `layout` must have been created, allocated or retrieved from the same `VkDevice`
- `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`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS COMPUTE

1.31.5 See Also

`vkCreatePipelineLayout`, `VkPipelineStageFlags`

1.32 vkCmdResetEvent(3)

1.32.1 Name

vkCmdResetEvent - Reset an event object to non-signaled state.

1.32.2 C Specification

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

1.32.3 Parameters

commandBuffer
The command buffer into which to insert the command.

event
The event object to reset to non-signaled state.

stageMask
Specifies when to reset the event (see `VkPipelineStageFlags` for more detail).

1.32.4 Description

`vkCmdResetEvent` causes the event object specified in *`event`* to be returned to the non-signaled state when the pipeline stages specified by *`stageMask`* have completed executing prior commands.

For definitions of the pipeline stages, see `VkPipelineStageFlags`.

Valid Usage

- *`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
- Each of *`commandBuffer`* and *`event`* must have been created, allocated or retrieved from the same `VkDevice`
- 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

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS COMPUTE

1.32.5 See Also

vkCmdSetEvent, vkSetEvent, vkResetEvent, VkPipelineStageFlags

1.33 vkCmdResetQueryPool(3)

1.33.1 Name

vkCmdResetQueryPool - Reset queries in a query pool.

1.33.2 C Specification

```
void vkCmdResetQueryPool(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount);
```

1.33.3 Parameters

commandBuffer
The command buffer into which the command is to be placed.

queryPool
The query pool containing the queries to be reset.

startQuery
The index of the first query to be reset.

queryCount
The number of queries to reset.

1.33.4 Description

vkCmdResetQueryPool resets *queryCount* starting at the entry index given by *startQuery* in the query pool specified by *queryPool*. The reset command is executed by the command buffer specified in *commandBuffer*. After execution, all queries are reset to inactive state and have zero values.

Valid Usage

- *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
- Each of *commandBuffer* and *queryPool* must have been created, allocated or retrieved from the same VkDevice
- *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*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS COMPUTE

1.33.5 See Also

vkCmdBeginQuery, vkCmdEndQuery, vkCreateQueryPool, vkDestroyQueryPool, vkGetQueryPoolResults, vkCmdCopyQueryPoolResults

1.34 vkCmdResolveImage(3)

1.34.1 Name

vkCmdResolveImage - Resolve regions of an image.

1.34.2 C Specification

```
void vkCmdResolveImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkImage                                     dstImage,
    VkImageLayout                               dstImageLayout,
    uint32_t                                    regionCount,
    const VkImageResolve*                       pRegions);
```

1.34.3 Parameters

commandBuffer
The command buffer into which the command is to be placed.

srcImage
The image that is the source of the resolve operation.

srcImageLayout
The layout of the source image at the time of the resolve.

dstImage
The image into which image data is to be resolved.

dstImageLayout
The layout of the destination image at the time of the resolve.

regionCount
The number of regions to resolve.

pRegions
An array of image regions to resolve.

1.34.4 Description

vkCmdResolveImage resolves regions of a source image into a destination image. The source and destination images are specified in *srcImage* and *dstImage*, respectively. The layout of the source and destination images must be provided in *srcImageLayout* and *dstImageLayout*, respectively. *pRegions* is a pointer to an array of *regionCount* VkImageResolve structures, the definition of each is:

```
typedef struct VkImageResolve {
    VkImageSubresourceLayers    srcSubresource;
    VkOffset3D                  srcOffset;
    VkImageSubresourceLayers    dstSubresource;
    VkOffset3D                  dstOffset;
    VkExtent3D                  extent;
} VkImageResolve;
```

Valid Usage

- *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*, *srcImage* and *dstImage* must have been created, allocated or retrieved from the same VkDevice
- 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 have a sample count equal to any valid sample count value other than VK_SAMPLE_COUNT_1_BIT
- *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 either of 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 either of 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**

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS

1.34.5 See Also

vkCmdBlitImage, vkCmdClearColorImage, vkCmdClearDepthStencilImage

1.35 vkCmdSetBlendConstants.txt(3)

1.35.1 Name

vkCmdSetBlendConstants - Set the values of blend constants.

1.35.2 C Specification

```
void vkCmdSetBlendConstants(
    VkCommandBuffer                             commandBuffer,
    const float                                 blendConstants[4]);
```

1.35.3 Parameters

commandBuffer
The command buffer into which to insert the command.

blendConstants
An array of values specifying the new blend constants.

1.35.4 Description

vkCmdSetBlendConstants sets the blend constants for the command buffer specified by *commandBuffer* to the values specified in the four element array *blendConstants*. Blend constants may be modified only if the current pipeline state object was created with the VK_DYNAMIC_STATE_BLEND_CONSTANTS dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_BLEND_CONSTANTS dynamic state enabled is first bound, the values of the blend constants are taken from the pipeline and attempts to change them using **vkCmdSetBlendConstants** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_BLEND_CONSTANTS dynamic state enabled is first bound, the current values of the blend constants become undefined and must be set using a call to **vkCmdSetBlendConstants**.

Valid Usage

- *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
- The currently bound graphics pipeline must have been created with the VK_DYNAMIC_STATE_BLEND_CONSTANTS dynamic state enabled

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.35.5 See Also

vkCreateGraphicsPipelines, VkPipelineDynamicStateCreateInfo

1.36 vkCmdSetDepthBias(3)

1.36.1 Name

vkCmdSetDepthBias - Set the depth bias dynamic state.

1.36.2 C Specification

```
void vkCmdSetDepthBias(
    VkCommandBuffer                             commandBuffer,
    float                                       depthBiasConstantFactor,
    float                                       depthBiasClamp,
    float                                       depthBiasSlopeFactor);
```

1.36.3 Parameters

commandBuffer
A handle to the command buffer into which to insert the command.

depthBiasConstantFactor
The constant bias factor.

depthBiasClamp
The bias clamp factor.

depthBiasSlopeFactor
The bias slope factor.

1.36.4 Description

vkCmdSetDepthBias sets the depth bias parameters for the command buffer specified by *commandBuffer*. The *depthBiasConstantFactor*, *depthBiasClamp* and *depthBiasSlopeFactor* parameters specify the new values for the depth bias calculation. The graphics pipeline bound to *commandBuffer* must have the VK_DYNAMIC_STATE_DEPTH_BIAS dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_DEPTH_BIAS dynamic state enabled is first bound, the values of the depth bias parameters are taken from the pipeline and attempts to change them using **vkCmdSetBlendConstants** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_DEPTH_BIAS dynamic state enabled is first bound, the current values of the depth bias parameters become undefined and must be set using a call to **vkCmdSetDepthBias**.

Valid Usage

- *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
- 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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.36.5 See Also

vkCreateGraphicsPipelines, VkPipelineDynamicStateCreateInfo

1.37 vkCmdSetDepthBounds(3)

1.37.1 Name

vkCmdSetDepthBounds - Set the depth bounds test values for a command buffer.

1.37.2 C Specification

```
void vkCmdSetDepthBounds(
    VkCommandBuffer                             commandBuffer,
    float                                       minDepthBounds,
    float                                       maxDepthBounds);
```

1.37.3 Parameters

commandBuffer
The command buffer into which to insert the command.

minDepthBounds
The minimum value for the depth bounds test range.

maxDepthBounds
The maximum value for the depth bounds test range.

1.37.4 Description

vkCmdSetDepthBounds sets the minimum and maximum values for the depth bounds test for the command buffer specified in *commandBuffer*. *minDepthBounds* and *maxDepthBounds* specify the minimum and maximum values for the depth bounds test respectively. A the value stored in the current depth attachment at a fragment's location lies between *minDepthBounds* and *maxDepthBounds*, then the depth bounds test passes, otherwise the test fails and the fragment's coverage bit is cleared.

The graphics pipeline bound to *commandBuffer* must have the `VK_DYNAMIC_STATE_DEPTH_BOUNDS` dynamic state enabled. When a pipeline that does not have `VK_DYNAMIC_STATE_DEPTH_BOUNDS` dynamic state enabled is first bound, the values of the depth bias parameters are taken from the pipeline and attempts to change them using **vkCmdSetBlendConstants** results in undefined behavior. When a pipeline does have `VK_DYNAMIC_STATE_DEPTH_BOUNDS` dynamic state enabled is first bound, the current values of the depth bias parameters become undefined and must be set using a call to **vkCmdSetDepthBias**.

If the depth bounds test for the current pipeline is not enabled, then it is as if the depth bounds test always passes and the values of *minDepthBounds* and *maxDepthBounds* are ignored.

The value of *maxDepthBounds* must be greater than or equal to the value of *minDepthBounds*.

Valid Usage

- *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
- 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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.37.5 See Also

vkCmdSetDepthBias, vkCreateGraphicsPipelines, VkPipelineDynamicStateCreateInfo

1.38 vkCmdSetEvent(3)

1.38.1 Name

vkCmdSetEvent - Set an event object to signaled state.

1.38.2 C Specification

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

1.38.3 Parameters

commandBuffer
The command buffer into which to insert the command.

event
The event object to set to signaled state.

stageMask
Specifies when the event becomes signaled (see `VkPipelineStageFlags` for more detail).

1.38.4 Description

`vkCmdSetEvent` causes the event object specified in *`event`* to be moved to the signaled state when the pipeline stages specified by *`stageMask`* have completed executing prior commands.

For definitions of the pipeline stages, see `VkPipelineStageFlags`.

Valid Usage

- *`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
- Each of *`commandBuffer`* and *`event`* must have been created, allocated or retrieved from the same `VkDevice`
- 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`

Host Synchronization

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

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	GRAPHICS COMPUTE

1.38.5 See Also

vkCmdResetEvent, vkSetEvent, vkResetEvent, VkPipelineStageFlags

1.39 vkCmdSetLineWidth(3)

1.39.1 Name

vkCmdSetLineWidth - Set the dynamic line width state.

1.39.2 C Specification

```
void vkCmdSetLineWidth(
    VkCommandBuffer                             commandBuffer,
    float                                       lineWidth);
```

1.39.3 Parameters

commandBuffer
The command buffer into which to insert the command.

lineWidth
The new line width.

1.39.4 Description

vkCmdSetLineWidth sets the dymamic line width for the command buffer specified in *commandBuffer* to the value specified in *lineWidth*. Line primitives drawn subsequent to this command, either directly using line topologies or by generation of line primitives mid-pipeline, will assume the specified width.

Dynamic line width may be modified only if the current pipeline state object was created with the VK_DYNAMIC_STATE_LINE_WIDTH dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_LINE_WIDTH dynamic state enabled is first bound, the line width is taken from the pipeline and attempts to change it using **vkCmdSetLineWidth** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_LINE_WIDTH dynamic state enabled is first bound, the current value for line width becomes undefined and must be set using a call to **vkCmdSetLineWidth**.

Valid Usage

- *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
- 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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.39.5 See Also

vkCmdSetDepthBias, vkCreateGraphicsPipelines, VkPipelineDynamicStateCreateInfo

1.40 vkCmdSetScissor(3)

1.40.1 Name

vkCmdSetScissor - Set the dynamic scissor rectangles on a command buffer.

1.40.2 C Specification

```
void vkCmdSetScissor(
    VkCommandBuffer                              commandBuffer,
    uint32_t                                     firstScissor,
    uint32_t                                     scissorCount,
    const VkRect2D*                              pScissors);
```

1.40.3 Parameters

commandBuffer
The command buffer into which to insert the command.

scissorCount
The number of scissor rectangles to update.

pScissors
A pointer to an array of structures defining the new scissor rectangles.

1.40.4 Description

vkCmdSetScissor sets the dynamic scissor state on the command buffer specified in *commandBuffer*. *scissorCount* specifies the number of scissor rectangles to update and pname;pScissors is pointer to an array of VkRect2D structures defining the new scissor rectangles. The definition of VkRect2D is:

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

The *offset* and *extent* members of VkRect2D specify the origin and size of the scissor rectangle, respectively. The rectangles numbered zero through *scissorCount* are updated and any remaining scissor rectangles become undefined.

The graphics pipeline bound to *commandBuffer* must have the VK_DYNAMIC_STATE_SCISSOR dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_SCISSOR dynamic state enabled is first bound, the origins and extents are taken from the pipeline and attempts to change them using **vkCmdSetScissor** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_SCISSOR dynamic state enabled is first bound, the current values of the scissor rectangle origins and extents become undefined and must be set using a call to **vkCmdSetScissor**.

Valid Usage

- `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`
- 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
- 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

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.40.5 See Also

`vkCreateGraphicsPipelines`, `VkPipelineDynamicStateCreateInfo`

1.41 vkCmdSetStencilCompareMask(3)

1.41.1 Name

vkCmdSetStencilCompareMask - Set the stencil compare mask dynamic state.

1.41.2 C Specification

```
void vkCmdSetStencilCompareMask(
    VkCommandBuffer                             commandBuffer,
    VkStencilFaceFlags                          faceMask,
    uint32_t                                    compareMask);
```

1.41.3 Parameters

commandBuffer
The command buffer into which to insert the command.

faceMask
The face or faces to which the new mask is to apply.

compareMask
The new value to use for the stencil compare mask.

1.41.4 Description

vkCmdSetStencilCompareMask sets the mask value used for stencil comparisons on the command buffer specified by *commandBuffer*. *faceMask* specifies the face or faces to which the new values are applied. It a bitmask comprised of members of the VkStencilFaceFlagBits enumeration, the definition of which is:

```
typedef enum VkStencilFaceFlagBits {
    VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
    VK_STENCIL_FACE_BACK_BIT = 0x00000002,
    VK_STENCIL_FRONT_AND_BACK = 0x00000003,
} VkStencilFaceFlagBits;
```

The graphics pipeline bound to *commandBuffer* must have the VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK dynamic state enabled is first bound, the value if the stencil compare mask is taken from the pipeline and attempts to change it using **vkCmdSetStencilCompareMask** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK dynamic state enabled is bound, the current value of the stencil compare mask becomes undefined and must be set using a call to **vkCmdSetStencilCompareMask**.

If the stencil test is disabled in the current graphics pipeline, then the value of the stencil compare mask is ignored.

Valid Usage

- *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
- The currently bound graphics pipeline must have been created with the VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK dynamic state enabled

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.41.5 See Also

vkCreateGraphicsPipelines, vkCmdSetStencilWriteMask, vkCmdSetStencilReference, VkPipelineDynamicStateCreateInfo

1.42 vkCmdSetStencilReference(3)

1.42.1 Name

vkCmdSetStencilReference - Set the stencil reference dynamic state.

1.42.2 C Specification

```
void vkCmdSetStencilReference(
    VkCommandBuffer                             commandBuffer,
    VkStencilFaceFlags                          faceMask,
    uint32_t                                    reference);
```

1.42.3 Parameters

commandBuffer
The command buffer into which to insert the command.

faceMask
The face or faces to which the command is to apply.

reference
The new value for the stencil reference dynamic state.

1.42.4 Description

vkCmdSetStencilReference sets the reference value used for stencil comparisons on the command buffer specified by *commandBuffer*. *faceMask* specifies the face or faces to which the new values are applied. It a bitmask comprised of members of the VkStencilFaceFlagBits enumeration, the definition of which is:

```
typedef enum VkStencilFaceFlagBits {
    VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
    VK_STENCIL_FACE_BACK_BIT = 0x00000002,
    VK_STENCIL_FRONT_AND_BACK = 0x00000003,
} VkStencilFaceFlagBits;
```

The graphics pipeline bound to *commandBuffer* must have the VK_DYNAMIC_STATE_STENCIL_REFERENCE dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_STENCIL_REFERENCE dynamic state enabled is first bound, the value if the stencil reference value is taken from the pipeline and attempts to change it using **vkCmdSetStencilReference** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_STENCIL_REFERENCE dynamic state enabled is bound, the current value of the stencil reference value becomes undefined and must be set using a call to **vkCmdSetStencilReference**.

If the stencil test is disabled in the current graphics pipeline, then the value of the stencil compare mask is ignored.

Valid Usage

- *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
- The currently bound graphics pipeline must have been created with the VK_DYNAMIC_STATE_STENCIL_REFERENCE dynamic state enabled

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.42.5 See Also

vkCreateGraphicsPipelines, vkCmdSetStencilCompareMask, vkCmdSetStencilWriteMask, VkPipelineDynamicStateCreateInfo

1.43 vkCmdSetStencilWriteMask(3)

1.43.1 Name

vkCmdSetStencilWriteMask - Set the stencil write mask dynamic state.

1.43.2 C Specification

```
void vkCmdSetStencilWriteMask(
    VkCommandBuffer                             commandBuffer,
    VkStencilFaceFlags                          faceMask,
    uint32_t                                    writeMask);
```

1.43.3 Parameters

commandBuffer
The command buffer into which to insert the command.

faceMask
The face or faces to which the new mask is to apply.

writeMask
The new value to use for the stencil compare mask.

1.43.4 Description

vkCmdSetStencilWriteMask sets the mask value used for stencil writes on the command buffer specified by *commandBuffer* to *writeMask*. *faceMask* specifies the face or faces to which the new values are applied. It a bitmask comprised of members of the VkStencilFaceFlagBits enumeration, the definition of which is:

```
typedef enum VkStencilFaceFlagBits {
    VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
    VK_STENCIL_FACE_BACK_BIT = 0x00000002,
    VK_STENCIL_FRONT_AND_BACK = 0x00000003,
} VkStencilFaceFlagBits;
```

The graphics pipeline bound to *commandBuffer* must have the VK_DYNAMIC_STATE_STENCIL_WRITE_MASK dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_STENCIL_WRITE_MASK dynamic state enabled is first bound, the value of the stencil write mask is taken from the pipeline and attempts to change it using **vkCmdSetStencilWriteMask** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_STENCIL_WRITE_MASK dynamic state enabled is bound, the current value of the stencil write mask becomes undefined and must be set using a call to **vkCmdSetStencilWriteMask**.

If the stencil test is disabled in the current graphics pipeline, then the value of the stencil write mask is ignored.

Valid Usage

- *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
- The currently bound graphics pipeline must have been created with the VK_DYNAMIC_STATE_STENCIL_WRITE_MASK dynamic state enabled

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.43.5 See Also

vkCreateGraphicsPipelines, vkCmdSetStencilCompareMask, vkCmdSetStencilReference, VkPipelineDynamicStateCreateInfo

1.44 vkCmdSetViewport(3)

1.44.1 Name

vkCmdSetViewport - Set the viewport on a command buffer.

1.44.2 C Specification

```
void vkCmdSetViewport(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstViewport,
    uint32_t                                    viewportCount,
    const VkViewport*                           pViewports);
```

1.44.3 Parameters

commandBuffer
The command buffer into which to insert the command.

viewportCount
The number of viewport rectangles to set.

pViewports
A pointer to an array of structures describing the viewports.

1.44.4 Description

vkCmdSetViewport sets the dynamic viewport state for the command buffer specified in *commandBuffer*. *viewportCount* is the number of viewports to update and *pViewports* is a pointer to an array of VkViewport structures describing the new viewport state. The definition of VkViewport is:

```
typedef struct VkViewport {
    float    x;
    float    y;
    float    width;
    float    height;
    float    minDepth;
    float    maxDepth;
} VkViewport;
```

The *x* and *y* members of VkViewport specifies the upper left corner of the viewport rectangle, in pixels. The *width* and *height* parameters specify the size of the rectangle, and are also expressed in pixels. The *minDepth* and *maxDepth* members specify the depth range for the viewport.

The viewports numbered zero through *viewportCount* are updated and any remaining viewports become undefined.

The graphics pipeline bound to *commandBuffer* must have the VK_DYNAMIC_STATE_VIEWPORT dynamic state enabled. When a pipeline that does not have VK_DYNAMIC_STATE_VIEWPORT dynamic state enabled is first bound, the origins and extents of the viewports are taken from the pipeline and attempts to change them using **vkCmdSetViewport** results in undefined behavior. When a pipeline does have VK_DYNAMIC_STATE_VIEWPORT dynamic state enabled is first bound, the current values of the viewport rectangle origins and extents become undefined and must be set using a call to **vkCmdSetViewport**.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pViewports* must be a pointer to an array of *viewportCount* valid VkViewport structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- *viewportCount* must be greater than 0
- 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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS

1.44.5 See Also

vkCreateGraphicsPipelines, VkPipelineDynamicStateCreateInfo

1.45 vkCmdUpdateBuffer(3)

1.45.1 Name

vkCmdUpdateBuffer - Update a buffer's contents from host memory.

1.45.2 C Specification

```
void vkCmdUpdateBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    dstBuffer,
    VkDeviceSize                                dstOffset,
    VkDeviceSize                                dataSize,
    const uint32_t*                             pData);
```

1.45.3 Parameters

commandBuffer
The command buffer into which the command is to be placed.

dstBuffer
The destination buffer.

dstOffset
The offset within *dstBuffer* where the data is to be placed.

dataSize
The size, in bytes of the data to be tranferred into the buffer.

pData
A pointer to the data to be transferred into the buffer.

1.45.4 Description

vkCmdUpdateBuffer updates the content of the buffer object specified in *dstBuffer* with the *dataSize* bytes of host memory sourced from *pData*. The data is placed at the offset specified by *dstOffset* into the buffer object.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *dstBuffer* must be a valid VkBuffer handle
- *pData* must be a pointer to an array of $\frac{dataSize}{4}$ uint32_t values
- *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
- Each of *commandBuffer* and *dstBuffer* must have been created, allocated or retrieved from the same VkDevice
- *dataSize* must be greater than 0
- *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
- *dstOffset* must be a multiple of 4
- *dataSize* must be less than or equal to 65536
- *dataSize* must be a multiple of 4

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Outside	TRANSFER GRAPHICS COMPUTE

1.45.5 See Also

vkCmdCopyBuffer

1.46 vkCmdWaitEvents(3)

1.46.1 Name

vkCmdWaitEvents - Wait for one or more events and insert a set of memory barriers.

1.46.2 C Specification

```
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);
```

1.46.3 Parameters

commandBuffer
: The command buffer in which to wait.

eventCount
: Number of event objects to wait on.

pEvents
: Array of *eventCount* number of event objects to wait on.

srcStageMask
: Mask of pipeline stages used to signal all of the events in *pEvents*.

dstStageMask
: Specifies which pipeline stages must wait for the events to become signaled (see `VkPipelineStageFlags` for more detail).

memoryBarrierCount
: Number of memory barriers to insert after waiting for the events.

ppMemoryBarriers
: Array of pointers to memory barrier structures specifying the parameters of the memory barriers to insert after waiting for the events. Each element of the array may point to a `VkMemoryBarrier`, `VkBufferMemoryBarrier`, or `VkImageMemoryBarrier` structure.

1.46.4 Description

vkCmdWaitEvents waits for a number of event objects to become signaled and inserts a set of memory barriers into the command buffer specified by *commandBuffer*.

vkCmdWaitEvents waits for each of the *eventCount* event object specified by *pEvents* to become signaled. The point at which each is signaled must have been specified in the command that caused the object to become signaled (either **vkSetEvent** or **vkCmdSetEvent**) and must also have the corresponding bit set in *srcStageMask*.

The *ppMemoryBarriers* parameter is a pointer to an array of *memoryBarrierCount* structures defining the parameters of memory barriers to insert after waiting for each of the events. Each element of the array may be an instance of `VkMemoryBarrier`, `VkBufferMemoryBarrier`, or `VkImageMemoryBarrier`, the definitions of each are, respectively:

```
typedef struct VkMemoryBarrier {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
} VkMemoryBarrier;
```

```
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;
```

```
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;
```

The memory barriers specified by *ppMemoryBarriers* cause writes as described by *outputMask* that were written by pipeline stages in *srcStageMask* prior to the wait to be made visible to reads as described by *inputMask* in pipeline stages in *dstStageMask* subsequent to the wait.

In case of global memory barriers inserted by passing an VkMemoryBarrier structure to the command prior writes in the requested pipeline stages to any memory location corresponding to the set of memory output coherency flags specified in the *outputMask* member of the structure are made coherent with subsequent reads in the requested pipeline stages of any memory location corresponding to the set of memory input coherency flags specified in the *inputMask* member of the structure.

In case of buffer memory barriers inserted by passing an VkBufferMemoryBarrier structure to the command prior writes in the requested pipeline stages to the specified sub-range of the buffer corresponding to the set of memory output coherency flags specified in the *outputMask* member of the structure are made coherent with subsequent reads in the requested pipeline stages of the specified sub-range of the buffer corresponding to the set of memory input coherency flags specified in the *inputMask* member of the structure.

In case of image memory barriers inserted by passing an VkImageMemoryBarrier structure to the command prior writes in the requested pipeline stages to the specified sub-range of the image corresponding to the set of memory output coherency flags specified in the *outputMask* member of the structure are made coherent with subsequent reads in the requested pipeline stages of the specified sub-range of the image corresponding to the set of memory input coherency flags specified in the *inputMask* member of the structure. Additionally, if the *oldLayout* and *newLayout* members of the structure don't match a layout transition is performed on the specified sub-range of the image as part of the memory barrier.

In case of buffer and image memory barriers the *srcQueueFamilyIndex* and *dstQueueFamilyIndex* members of the corresponding memory barrier structures can specify the parameters of a transfer of ownership between two distinct families of queues of a shared buffer or image object created with the VK_SHARING_MODE_EXCLUSIVE sharing mode. In case of regular resource transitions both *srcQueueFamilyIndex* and *dstQueueFamilyIndex* should be set to VK_QUEUE_FAMILY_IGNORED to indicate no transfer of ownership between queue families. In case of resource transitions involving ownership transfer of shared buffers or images one of these two members have to match the queue family index the command buffer specified by `commandBuffer` was created for, while the other should specify the queue family index the ownership transfer is released to or acquired from. Ownership transferring resource transitions have to be performed both on a queue from the source queue family and on a queue from the destination queue family (see `VkSharingMode` for more detail).

If *inputMask* is zero in any of the memory barrier structures then prior writes will only be coherent with any type of subsequent read after a future resource transition command specifies a non-empty set of memory input coherency control flags. This allows flushing device output caches unconditionally.

If *outputMask* is zero in any of the memory barrier structures then subsequent reads will only be coherent with any type of prior write if an earlier resource transition command specified a non-empty set of memory output coherency control flags. This allows invalidating device input caches unconditionally.

Valid Usage

- *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`
- Each of *commandBuffer* and the elements of *pEvents* must have been created, allocated or retrieved from the same `VkDevice`
- *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

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS COMPUTE

1.46.5 See Also

vkCmdSetEvent, vkCmdResetEvent, vkSetEvent, vkResetEvent, vkCmdPipelineBarrier, VkMemoryBarrier, VkBufferMemoryBarrier, VkImageMemoryBarrier, VkPipelineStageFlagBits

1.47 vkCmdWriteTimestamp(3)

1.47.1 Name

vkCmdWriteTimestamp - Write a device timestamp into a query object.

1.47.2 C Specification

```
void vkCmdWriteTimestamp(
    VkCommandBuffer                             commandBuffer,
    VkPipelineStageFlagBits                     pipelineStage,
    VkQueryPool                                 queryPool,
    uint32_t                                    query);
```

1.47.3 Parameters

commandBuffer
The command buffer into which the command will be placed.

pipelineStage
The stage of the pipeline at which the timestamp will be written.

queryPool
A handle to the query pool object containing the query.

entry
The entry in the query pool at which to write the query.

1.47.4 Description

vkCmdWriteTimestamp places a command into the command buffer specified by *commandBuffer* which, when executed, will cause the GPU to write its internal timestamp into the query pool specified by *queryPool* at the entry specified in *entry*. The timestamp is written when the command passes the pipeline stage specified by *pipelineStage*. The pipeline stage is a single member of the VkPipelineStageFlagBits enumeration, the definition of which is:

```
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,
} VkPipelineStageFlagBits;
```

If an implementation is not capable of writing a timestamp value at the pipeline point specified, it may at its option write the timestamp at any point appearing later in the logical pipeline. However, it must do this consistently for similar pipeline configurations.

Valid Usage

- *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
- Each of *commandBuffer* and *queryPool* must have been created, allocated or retrieved from the same `VkDevice`
- The query identified by *queryPool* and *query* must be *unavailable*
- The command pool's queue family must support a non-zero *timestampValidBits*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command Properties

Command Buffer Levels	Render Pass Scope	Supported Queue Types
Primary Secondary	Both	GRAPHICS COMPUTE

1.47.5 See Also

`vkCmdSetEvent`

1.48 vkCreateBuffer(3)

1.48.1 Name

vkCreateBuffer - Create a new buffer object.

1.48.2 C Specification

```
VkResult vkCreateBuffer(
    VkDevice                                    device,
    const VkBufferCreateInfo*                   pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkBuffer*                                   pBuffer);
```

1.48.3 Parameters

device
The device with which to create the new buffer object.

pCreateInfo
Pointer to data structure containing information about the object to be created.

pBuffer
Pointer to a variable to receive a handle to the new buffer object.

1.48.4 Description

vkCreateBuffer creates a new buffer object using the device specified in *device*. The resulting buffer object handle is written into the variable whose address is given in *pBuffer*. *pCreateInfo* is a pointer to a data structure describing the buffer to be created and is of type VkBufferCreateInfo, whose definition is:

```
typedef struct VkBufferCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkBufferCreateFlags      flags;
    VkDeviceSize             size;
    VkBufferUsageFlags       usage;
    VkSharingMode            sharingMode;
    uint32_t                 queueFamilyIndexCount;
    const uint32_t*          pQueueFamilyIndices;
} VkBufferCreateInfo;
```

Valid Usage

- *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
- 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*

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

1.48.5 See Also

vkCreateImage, vkCreateBufferView

1.49 vkCreateBufferView(3)

1.49.1 Name

vkCreateBufferView - Create a new buffer view object.

1.49.2 C Specification

```
VkResult vkCreateBufferView(
    VkDevice                                    device,
    const VkBufferViewCreateInfo*               pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkBufferView*                               pView);
```

1.49.3 Parameters

device
: The device with which to create the buffer view.

pCreateInfo
: A pointer to a structure containing information to be placed in the object.

pView
: A pointer to a variable which will receive the handle to the new object.

1.49.4 Description

vkCreateBufferView creates a new buffer view using the information contained in *pCreateInfo* and the device specified in *device*. Upon success, a handle to the new view object is deposited in the variable pointed to by *pView*. *pCreateInfo* should point to an instance of the VkBufferViewCreateInfo structure, the definition of which is:

```
typedef struct VkBufferViewCreateInfo {
    VkStructureType            sType;
    const void*                pNext;
    VkBufferViewCreateFlags    flags;
    VkBuffer                   buffer;
    VkFormat                   format;
    VkDeviceSize               offset;
    VkDeviceSize               range;
} VkBufferViewCreateInfo;
```

Valid Usage

- *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

1.49.5 See Also

vkCreateBufferView

1.50 vkCreateCommandPool(3)

1.50.1 Name

vkCreateCommandPool - Create a new command pool object.

1.50.2 C Specification

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

1.50.3 Parameters

device
: The device with which to create the command pool.

pCreateInfo
: A pointer to a structure containing information about the command pool.

pCommandPool
: The address of a variable to receive the handle to the new command pool.

1.50.4 Description

vkCreateCommandPool creates a new command pool object using *device* and places its handle in the variable whose address is given in *pCommandPool*. *pCreateInfo* is a pointer to an instance of the VkCommandPoolCreateInfo structure which contains information about how to create the new command pool. Its definition is:

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

queueFamilyIndex indicates the family of queues which the command buffer can be submitted to, as well as the subset of commands which may be recorded on it.

flags is a bitfield of flags indicating usage behavior for the pool and command buffers allocated from it. Possible values include:

VK_COMMAND_POOL_CREATE_TRANSIENT_BIT indicates that command buffers created from the pool will be short-lived, meaning that they will be reset or destroyed in a relatively short timeframe.

VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT controls whether it is legal to call **vkResetCommandBuffer** on a command buffer allocated from the pool. If this is not set, then the command buffers may only be reset in bulk by calling **vkResetCommandPool**.

Valid Usage

- *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

1.50.5 See Also

vkDestroyCommandPool, vkResetCommandPool

1.51 vkCreateComputePipelines(3)

1.51.1 Name

vkCreateComputePipelines - Creates a new compute pipeline object.

1.51.2 C Specification

```
VkResult vkCreateComputePipelines(
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    uint32_t                                    createInfoCount,
    const VkComputePipelineCreateInfo*          pCreateInfos,
    const VkAllocationCallbacks*                pAllocator,
    VkPipeline*                                 pPipelines);
```

1.51.3 Parameters

device
A handle to the device to use to create the new compute pipelines.

pipelineCache
A handle to a pipeline cache from which the result of previous compiles may be retrieved, and to which the result of this compile may be stored.

createInfoCount
The number of pipelines to create.

pCreateInfos
Pointer to an array of *createInfoCount* VkComputePipelineCreateInfo structures defining the contents of the new pipelines.

pPipelines
A pointer to an array to receive the handles to the new compute pipeline objects.

1.51.4 Description

vkCreateComputePipelines creates new compute pipeline objects using the device specified in *device* and the creation information specified in the structures pointed to by *pCreateInfos* and deposits the resulting handles in the array pointed to by *pPipelines*. The definition of VkComputePipelineCreateInfo is:

```
typedef struct VkComputePipelineCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineCreateFlags               flags;
    VkPipelineShaderStageCreateInfo     stage;
    VkPipelineLayout                    layout;
    VkPipeline                          basePipelineHandle;
    int32_t                             basePipelineIndex;
} VkComputePipelineCreateInfo;
```

CREATE INFO DETAILS

- *sType* indicates the type of this structure and must be VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO.
- *pNext* is a pointer to an extension-specific structure (can be NULL).

- `stage` is a `VkPipelineShaderStageCreateInfo` describing the compute shader.
- `flags` controls how the driver will create the pipeline.
- `layout` the description of binding locations used by both the pipeline and the descritor sets.
- `basePipelineHandle` the pipeline to derive from (can be **VK_NULL_HANDLE**, if pipeline is not derived).
- `basePipelineIndex` the index into the `pCreateInfos` parameter to **vkCreateComputePipelines**.

The parameters `basePipelineHandle` and `basePipelineIndex` are ignored unless `flags` has the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` bit set. If using the `basePipelineIndex` parameter, the index must refer to a `pCreateInfos` parameter passed to vkCreateComputePipelines that appeared earlier than the current `VkComputePipelineCreateInfo` in the list. The parameters `basePipelineHandle` and `basePipelineIndex` are mutually exclusive. If you specify a valid `basePipelineHandle`, `basePipelineIndex` must be set to -1. If you specify a valid `basePipelineIndex`, `basePipelineHandle` must be **VK_NULL_HANDLE**.

Valid Usage

- `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`
- Each of `device` and `pipelineCache` that are valid handles must have been created, allocated or retrieved from the same `VkPhysicalDevice`
- 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

Return Codes

Success

- `VK_SUCCESS`

Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

1.51.5 See Also

`vkCreateGraphicsPipelines`, `vkCmdBindPipeline`

1.52 vkCreateDescriptorPool(3)

1.52.1 Name

vkCreateDescriptorPool - Creates a descriptor pool object.

1.52.2 C Specification

```
VkResult vkCreateDescriptorPool(
    VkDevice                                    device,
    const VkDescriptorPoolCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorPool*                           pDescriptorPool);
```

1.52.3 Parameters

device
Logical device which will own the new descriptor pool object.

pCreateInfo
A pointer to a structure containing parameters of the new pool object.

pDescriptorPool
Pointer to a variable which will receive a handle to the new descriptor pool object.

1.52.4 Description

vkCreateDescriptorPool creates a new descriptor pool object using *device*. Descriptor sets may be allocated from the resulting descriptor pool object by calling vkAllocateDescriptorSets. *pCreateInfo* is a pointer to an insance of the VkDescriptorPoolCreateInfo structure containing parameters describing the new pool object. The definition of VkDescriptorPoolCreateInfo is:

```
typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDescriptorPoolCreateFlags     flags;
    uint32_t                        maxSets;
    uint32_t                        poolSizeCount;
    const VkDescriptorPoolSize*     pPoolSizes;
} VkDescriptorPoolCreateInfo;
```

The *sType* member of the *pCreateInfo* structure should be VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO. The *pNext* member is reserved for use by extensions and should be set to **NULL**.

The *flags* member of VkDescriptorPoolCreateInfo is a set of flags describing the intended usage of the pool and is formed from members of the VkDescriptorPoolCreateFlagBits enumeration, the definition of which is:

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

If VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT is set in *flags* then descriptor sets allocated from the pool may be returned to the pool by calling vkFreeDescriptorSets. If this flag is clear then individual sets allocated from the pool may not be returned to the pool and are considered allocated until vkResetDescriptorPool is called on the pool object.

The *maxSets* member specifies the maximum number of descriptor sets that will be allocated from the pool. *pPoolSizes* is a pointer to an array of *poolSizeCount* VkDescriptorPoolSize structures, each describing a type of descriptor and the number of that type of descriptor to be included in the pool. The definition of the VkDescriptorPoolSize structure is:

```
typedef struct VkDescriptorPoolSize {
    VkDescriptorType    type;
    uint32_t            descriptorCount;
} VkDescriptorPoolSize;
```

Each element of the *pPoolSizes* array specifies a type of descriptor in *type* and the count of that type of descriptor in *descriptorCount*. The *type* member must be a member of the VkDescriptorType enumeration, the definition of which is:

```
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;
```

Upon success, a handle to the newly created descriptor pool is placed in the variable whose address is specified in *pDescriptorPool*.

Valid Usage

- *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

1.52.5 See Also

vkAllocateDescriptorSets, vkFreeDescriptorSets, vkResetDescriptorPool, vkDestroyDescriptorPool

1.53 vkCreateDescriptorSetLayout(3)

1.53.1 Name

vkCreateDescriptorSetLayout - Create a new descriptor set layout.

1.53.2 C Specification

```
VkResult vkCreateDescriptorSetLayout(
    VkDevice                                    device,
    const VkDescriptorSetLayoutCreateInfo*      pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorSetLayout*                      pSetLayout);
```

1.53.3 Parameters

device
The device with which to create the layout object.

pCreateInfo
Pointer to a structure specifying information to be placed in the object

pSetLayout
Pointer to a variable which will receive the new handle.

1.53.4 Description

vkCreateDescriptorSetLayout creates a new descriptor set layout usable by the device specified in *device* using the information contained in the structure pointed to by *pCreateInfo*. If successful, a handle to the newly created layout object is placed in the variable pointed to by *pSetLayout*. The description of the layout is specified in *pCreateInfo*, which is a pointer to an instance of the VkDescriptorSetLayoutCreateInfo structure, the definition of which is:

```
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType                         sType;
    const void*                             pNext;
    VkDescriptorSetLayoutCreateFlags        flags;
    uint32_t                                bindingCount;
    const VkDescriptorSetLayoutBinding*     pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

The *bindingCount* member of *pCreateInfo* specifies the number of bindings contained in the set. This is the number of elements in the array pointed to by *pBinding*, which is an array of VkDescriptorSetLayoutBinding structures. The definition of VkDescriptorSetLayoutBinding is:

```
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t              binding;
    VkDescriptorType      descriptorType;
    uint32_t              descriptorCount;
    VkShaderStageFlags    stageFlags;
    const VkSampler*      pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

Each element of the *pBinding* array specifies a descriptor or an array of descriptors to be included in the set layout. *descriptorType* contains the descriptor type and must be one of the VkDescriptorType enumerants, the complete list of which is:

```
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;
```

The `stageFlags` member specifies which pipeline shader stages may access the resource. This is a bitwise combination of the `VkShaderStageFlags` enumerant, the list of which is:

```
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;
```

If a shader stage is not included in `stageFlags`, then the resource may not be accessed from that stage within any pipeline using the set layout.

If `descriptorType` member specifies a `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` type descriptor, then the `pImmutableSamplers` member may 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 `arraySize` 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.

Valid Usage

- `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`

1.53.5 See Also

vkAllocateDescriptorSets, vkFreeDescriptorSets, vkCreateDescriptorPool

1.54 vkCreateDevice(3)

1.54.1 Name

vkCreateDevice - Create a new device instance.

1.54.2 C Specification

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

1.54.3 Parameters

physicalDevice
Handle to the physical device upon which to create the logical device.

pCreateInfo
Pointer to a structure containing creation info.

pDevice
Pointer to a variable to receive the handle to the new device instance.

1.54.4 Description

vkCreateDevice creates a new device instance on the physical device specified by *physicalDevice* and places the resulting device handle in the variable pointed to by *pDevice*. Information about how the device should be created is passed in an instance of VkDeviceCreateInfo whose address is passed in *pCreateInfo*. The definition of VkDeviceCreateInfo is:

```
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;
```

Valid Usage

- *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_LAYER_NOT_PRESENT
- VK_ERROR_EXTENSION_NOT_PRESENT
- VK_ERROR_FEATURE_NOT_PRESENT
- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_DEVICE_LOST

- *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.

1.54.5 See Also

vkDestroyDevice

1.55 vkCreateEvent(3)

1.55.1 Name

vkCreateEvent - Create a new event object.

1.55.2 C Specification

```
VkResult vkCreateEvent(
    VkDevice                                    device,
    const VkEventCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkEvent*                                    pEvent);
```

1.55.3 Parameters

device
A handle to the device with which to create the event.

pCreateInfo
A pointer to the creation info structure.

pEvent
The address of an VK_EVENT variable that will receive the handle to the new event.

1.55.4 Description

vkCreateEvent creates a new event object using the device specified as *device*. A handle to the newly created event object is placed the variable pointed to by *pEvent*. *pCreateInfo* is a pointer to an instance of a VkEventCreateInfo structure containing information about the state in which to create the new object. The definition of VkEventCreateInfo is:

```
typedef struct VkEventCreateInfo {
    VkStructureType       sType;
    const void*           pNext;
    VkEventCreateFlags    flags;
} VkEventCreateInfo;
```

The *sType* member of the VkEventCreateInfo structure should be set to VK_STRUCTURE_TYPE_EVENT_CREATE_INFO. The *pNext* member is reserved for use by extensions and should be set to **NULL**.

The *flags* member specifies additional information about the event to be created. There are presently no flags defined for this member and it should be set to zero.

Valid Usage

- *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

1.55.5 See Also

vkSetEvent, vkResetEvent, vkCmdSetEvent, vkCmdResetEvent

1.56 vkCreateFence(3)

1.56.1 Name

vkCreateFence - Create a new fence object.

1.56.2 C Specification

```
VkResult vkCreateFence(
    VkDevice                                    device,
    const VkFenceCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFence*                                    pFence);
```

1.56.3 Parameters

device
A handle to the device with which to create the fence.

pCreateInfo
A pointer to a structure containing information about how to create the fence.

pFence
A pointer to a variable to receive the handle to the newly created fence object.

1.56.4 Description

vkCreateFence creates a new fence object using the device specified by *device* and places the resulting object handle in the variable pointed to by *pFence*. Information about how the fence should be created is passed in an instance of VkFenceCreateInfo whose address is given in *pCreateInfo*. The definition of VkFenceCreateInfo is:

```
typedef struct VkFenceCreateInfo {
    VkStructureType       sType;
    const void*           pNext;
    VkFenceCreateFlags    flags;
} VkFenceCreateInfo;
```

The *sType* member of the VkFenceCreateInfo structure should be set to VK_STRUCTURE_TYPE_FENCE_CREATE_INFO. The *pNext* member is reserved for use by extensions and should be set to **NULL**.

The *flags* member specifies additional information about the fence to be created. It is a bitfield made up from the members of the VkFenceCreateFlagBits enumeration, the definition of which is:

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

If VK_FENCE_CREATE_SIGNALED_BIT is set then the fence is created already signaled, otherwise, the fence is created in an unsignaled state.

A fence becomes signaled when it is submitted to a queue with a call to vkQueueSubmit. A fence may be reset to unsignaled state with a call to vkResetFences.

Valid Usage

- *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

1.56.5 See Also

vkWaitForFences, vkDestroyFence, vkResetFences, vkQueueSubmit

1.57 vkCreateFramebuffer(3)

1.57.1 Name

vkCreateFramebuffer - Create a new framebuffer object.

1.57.2 C Specification

```
VkResult vkCreateFramebuffer(
    VkDevice                                    device,
    const VkFramebufferCreateInfo*              pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFramebuffer*                              pFramebuffer);
```

1.57.3 Parameters

device
The device with which to create the framebuffer object.

pCreateInfo
A pointer to a structure containing information about how to create the object.

pFramebuffer
A pointer to a variable which will receive the handle to the new object.

1.57.4 Description

vkCreateFramebuffer creates a new framebuffer object using the information contained in *pCreateInfo* and the device specified in *device*. Upon success, a handle to the new framebuffer object is deposited in the variable pointed to by *pFramebuffer*. *pCreateInfo* should point to an instance of the VkFramebufferCreateInfo structure, the definition of which is:

```
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;
```

The attachments in *pAttachments* correspond in order to the attachment descriptions in the *renderPass*. The attachment view must have the same format, sample count, and initial layout as the render pass's attachment description. All attachment views must also have dimensions at least as large as the framebuffer's *width*, *height*, and *layers*.

The framebuffer may be used in combination with any render pass that has the same attachment count, and corresponding attachments have the same format and sample count.

Valid Usage

- *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

1.57.5 See Also

vkCmdBeginRenderPass, vkDestroyFramebuffer

1.58 vkCreateGraphicsPipelines(3)

1.58.1 Name

vkCreateGraphicsPipelines - Create graphics pipelines.

1.58.2 C Specification

```
VkResult vkCreateGraphicsPipelines(
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    uint32_t                                    createInfoCount,
    const VkGraphicsPipelineCreateInfo*         pCreateInfos,
    const VkAllocationCallbacks*                pAllocator,
    VkPipeline*                                 pPipelines);
```

1.58.3 Parameters

device
A handle to the device to use to create the new graphics pipeline(s).

pipelineCache
A handle to a pipeline cache from which the result of previous compiles may be retrieved, and to which the result of this compile may be stored.

createInfoCount
The number of pipelines to create.

pCreateInfos
Pointer to an array of *createInfoCount* VkGraphicsPipelineCreateInfo structures defining the contents of the new pipelines.

pPipelines
A pointer to an array to receive the handle(s) to the new graphics pipeline object(s).

1.58.4 Description

vkCreateGraphicsPipelines creates new graphics pipeline objects using the device specified in *device* and the creation information specified in the structures pointed to by *pCreateInfos* and deposits the resulting handles in the array pointed to by *pPipelines*. The definition of VkGraphicsPipelineCreateInfo is:

```
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;
```

Create Info Details

- *pStages* points to an array of *stageCount* VkPipelineShaderStageCreateInfo objects describing the stages comprising the pipeline. At minimum, the vertex shader stage must be defined.

- *pVertexInputState* points to a VkPipelineVertexInputStateCreateInfo object describing the layout of the vertex buffers as well as the attributes within the buffers for the pipeline.

- *pInputAssemblyState* points to a VkPipelineInputAssemblyStateCreateInfo object describing the input assembly state for the pipeline, including the primitive type and topology.

- *pTessellationState* points to a VkPipelineTessellationStateCreateInfo object describing the patch control state for the tesselation stage of the pipeline, or is set to *NULL* if no tesselation stage is defined.

- *pViewportState* points to a VkPipelineViewportStateCreateInfo object describing the viewport state for the pipeline.

- *pRasterizationState* points to a VkPipelineRasterizationStateCreateInfo object describing the rasterizer state for the pipeline, including fill mode, clip mode, and face orientation.

- *pMultisampleState* points to a VkPipelineMultisampleStateCreateInfo object describing the multisample state for the pipeline.

- *pDepthStencilState* points to a VkPipelineDepthStencilStateCreateInfo object describing the depth and stencil state for the pipeline.

- *pColorBlendState* points to a VkPipelineColorBlendStateCreateInfo object describing the color buffer state for the pipeline, including state for each of the attachments that will be bound to the framebuffer.

- *flags* is an instance of VkPipelineCreateFlags, indicating additional usage hint (e.g., if this pipeline will be used to create derivative pipelines).

- *layout* is a handle to a VkPipelineLayout object created with **vkCreatePipelineLayout**.

- *renderPass* is a handle to a VkRenderPass object describing a renderpass the pipeline will be compatible with.

- *subpass* is the index of the subpass in the *renderPass* the pipeline will be compatible with.

The created pipeline may only be used in a subpass compatible with the provided *renderPass* and *subpass*. Two subpasses are compatible if they have the same index in their render passes, and if the render pass descriptions are identical except for attachment load and store ops and image layouts. For a render pass with only one subpass, the subpasses are compatible if they have the same number and kind of attachments, and if corresponding attachments have the same format and sample count.

Valid Usage

- *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*
- Each of *device* and *pipelineCache* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

1.58.5 See Also

vkCreateComputePipelines, vkCmdBindPipeline, vkDestroyPipeline

1.59 vkCreateImage(3)

1.59.1 Name

vkCreateImage - Create a new image object.

1.59.2 C Specification

```
VkResult vkCreateImage(
    VkDevice                                    device,
    const VkImageCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkImage*                                    pImage);
```

1.59.3 Parameters

device
A handle to the device with which to create the image.

pCreateInfo
A pointer to a `VkImageCreateInfo` structure specifying the properties of the new image.

pImage
A pointer to a variable to receive the handle to the resulting image.

1.59.4 Description

`vkCreateImage` creates a new image object and places the resulting handle in the variable pointed to by *`pImage`*. The properties of the new image are specified in an instance of a `VkImageCreateInfo` structure whose address is given in *`pCreateInfo`*. The definition of `VkImageCreateInfo` is:

```
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;
```

Valid Usage

- `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
- 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`

Return Codes

Success

- `VK_SUCCESS`

Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

1.59.5 See Also

`vkCreateSampler`

1.60 vkCreateImageView(3)

1.60.1 Name

vkCreateImageView - Create an image view from an existing image.

1.60.2 C Specification

```
VkResult vkCreateImageView(
    VkDevice                                    device,
    const VkImageViewCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkImageView*                                pView);
```

1.60.3 Parameters

device
Logical device which owns the image.

pCreateInfo
Specifies properties of the new view.

pView
Returns the requested object.

1.60.4 Description

vkCreateImageView creates a new view of a source image in a compatible format, allowing casting of image data from one format to another. Image views may be bound into descriptor sets to allow them to be accessed in shaders, or be bound as color attachments. *device* specifies the device that is to be used to create the new view. *pCreateInfo* is a pointer to an instance of the VkImageViewCreateInfo structure defining the properties of the new view object. The definition of VkImageViewCreateInfo is:

```
typedef struct VkImageViewCreateInfo {
    VkStructureType            sType;
    const void*                pNext;
    VkImageViewCreateFlags     flags;
    VkImage                    image;
    VkImageViewType            viewType;
    VkFormat                   format;
    VkComponentMapping         components;
    VkImageSubresourceRange    subresourceRange;
} VkImageViewCreateInfo;
```

The *sType* member of *pCreateInfo* should be VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO. The *image* member contains the handle to the parent object of which to create a view. *viewType* specifies the type of view to be created and should be a member of the VkImageViewType enumeration, the definition of which is:

```
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 `format` member of `pCreateInfo` specifies the image format for the newly created view and should be compatible with the base format of the parent image specified in `image`. The `components` member is an instance of the `VkComponentMapping` structure which defines component ordering for data read from the view. The `subresourceRange` member of the `pCreateInfo` specifies the range of the parent resource to be visible through the new view.

The `flags` member of `pCreateInfo` is reserved and must be 0.

Valid Usage

- `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`

1.60.5 See Also

`vkCreateImage`, `vkCreateBuffer`, `vkCreateBufferView`

1.61 vkCreateInstance(3)

1.61.1 Name

vkCreateInstance - Create a new Vulkan instance

1.61.2 C Specification

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

1.61.3 Parameters

pCreateInfo
Pointer to instance creation structure.

pInstance
Pointer to variable which will receive the new instance handle.

1.61.4 Description

vkCreateInstance creates a new Vulkan instance and places a handle to it in the variable pointed to by *pInstance*. *pCreateInfo* is a pointer to an instance of the VkInstanceCreateInfo structure containing information about how the instance should be created. The definition of VkInstanceCreateInfo is:

```
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;
```

The *sType* member of VkInstanceCreateInfo should be set to VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO. The *pNext* member of VkInstanceCreateInfo is reserved for use by extensions and should be set to **NULL**.

The *pApplicationInfo* member, if non-**NULL**, points to an instance of the VkApplicationInfo structure containing information about the application. The expected contents of the *pApplicationInfo* member are documented below.

The *enabledLayerNameCount* member of VkInstanceCreateInfo specifies the number of global layers to enable, and *ppEnabledLayerNames* is a pointer to an array of *enabledLayerNameCount* **NULL**-terminated UTF-8 strings containing the names of layers that should be enabled globally. If *enabledLayerNameCount* is zero, then *ppEnabledLayerNames* is ignored and no global layers are enabled. If a layer cannot be found, the call will not create a VkInstance and will return VK_ERROR_LAYER_NOT_PRESENT.

Similary, information about global extensions is specified in the *enabledExtensionNameCount* and *ppEnabledExtensionNames* members. *enabledExtensionNameCount* specifies the number of global extensions to enable and *ppEnabledExtensionNames* is a pointer to an array of pointers to **NULL**-terminated UTF-8 strings containing the extension names. If an extension is provided by a layer, both the layer and extension must be specified at **vkCreateInstance** time. If a extension cannot be found, the call will not create a VkInstance and will return VK_ERROR_LAYER_NOT_PRESENT.

If *enabledExtensionNameCount* is zero then no extensions are enabled and *ppEnabledExtensionNames* is ignored.

The definition of the VkApplicationInfo structure is as follows:

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

The `sType` member of `VkApplicationInfo` should be set to `VK_STRUCTURE_TYPE_APPLICATION_INFO`. The `pNext` member of `VkApplicationInfo` is reserved for use by extensions and should be set to **NULL**.

`pApplicationName` is a pointer to a **NULL**-terminated UTF-8 string containing the name of the application. `applicationVersion` contains an application-specific version number. It is recommended that new versions of an existing application specify monotonically increasing values for `applicationVersion`.

If the application is built on a reusable engine, the name of the engine may be specified in the NULL-terminated UTF-8 string pointed to by `pEngineName`. `engineVersion` is the version of the engine used to create the application.

Finally, `apiVersion` is the version of the Vulkan API that the application expects to use.

Any application memory required by the instance will be allocated by calling functions specified in the structure pointed to by `pAllocCb`. The definition of `VkAllocationCallbacks` is:

```
typedef struct VkAllocationCallbacks {
    void*                                   pUserData;
    PFN_vkAllocationFunction                pfnAllocation;
    PFN_vkReallocationFunction              pfnReallocation;
    PFN_vkFreeFunction                      pfnFree;
    PFN_vkInternalAllocationNotification    pfnInternalAllocation;
    PFN_vkInternalFreeNotification          pfnInternalFree;
} VkAllocationCallbacks;
```

The `VkAllocationCallbacks` structure contains two function pointers. `pfnAllocation` points to an allocation function whose protoype should be of the following form:

```
typedef void* (*PFN_vkAllocationFunction)(
    void*                                       pUserData,
    size_t                                      size,
    size_t                                      alignment,
    VkSystemAllocationScope                     allocationScope);
```

`pUserData` is set to the value of `pUserData` in the allocation info structure passed to **`vkCreateInstance`**. `size` is the size of the desired allocation, `alignment` is the desired allocation, in bytes, and `allocationScope` represents the intended usage of the allocation. The return value of function is a pointer to the newly allocated memory.

The `pfnFree` member of `VkAllocationCallbacks` points to an instance of the following function:

```
typedef void (*PFN_vkFreeFunction)(
    void*                                       pUserData,
    void*                                       pMemory);
```

Again, the `pUserData` parameter is initialized to the value passed in the `VkAllocationCallbacks` structure passed to **`vkCreateInstance`**. `pMemory` is a pointer to the memory to be freed.

Valid Usage

- `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

1.61.5 See Also

vkDestroyInstance

1.62 vkCreatePipelineCache(3)

1.62.1 Name

vkCreatePipelineCache - Creates a new pipeline cache

1.62.2 C Specification

```
VkResult vkCreatePipelineCache(
    VkDevice                                    device,
    const VkPipelineCacheCreateInfo*            pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkPipelineCache*                            pPipelineCache);
```

1.62.3 Parameters

device
A handle to the device that will create the pipeline cache.

pCreateInfo
A pointer to a `VkPipelineCacheCreateInfo` object describing the pipeline cache to be created.

pPipelineCache
A pointer that will receive the handle to the newly created pipeline cache.

1.62.4 Description

```
typedef struct VkPipelineCacheCreateInfo {
    VkStructureType               sType;
    const void*                   pNext;
    VkPipelineCacheCreateFlags    flags;
    size_t                        initialDataSize;
    const void*                   pInitialData;
} VkPipelineCacheCreateInfo;
```

CREATE INFO DETAILS

- *initialDataSize* is the size of the initial data to populate the cache.
- *pInitialData* is a pointer to the initial data to populate the cache.
- *maxSize* specifies an upper bound on the size the cache will grow to, with -1 indicating that the cache may grow without bound.

Valid Usage

- *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

1.62.5 See Also

vkCreateGraphicsPipelines, vkCreateComputePipelines, vkGetPipelineCacheData, vkMergePipelineCaches

1.63 vkCreatePipelineLayout(3)

1.63.1 Name

vkCreatePipelineLayout - Creates a new pipeline layout object.

1.63.2 C Specification

```
VkResult vkCreatePipelineLayout(
    VkDevice                                    device,
    const VkPipelineLayoutCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkPipelineLayout*                           pPipelineLayout);
```

1.63.3 Parameters

device
The device with which to create the new pipeline layout object.

pCreateInfo
A pointer to structure specifying the properties of the new pipeline layout.

pPipelineLayout
Pointer to a variable to receive a handle to the new pipeline layout object.

1.63.4 Description

vkCreatePipelineLayout creates a new pipeline layout object for the device specified in *device*. The resulting pipeline layout object handle is written into the variable whose address is given in *pPipelineLayout*.

pCreateInfo is a pointer to an instance of a VkPipelineLayoutCreateInfo structure describing the new pipeline layout. The definition of VkPipelineLayoutCreateInfo is:

```
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkPipelineLayoutCreateFlags     flags;
    uint32_t                        setLayoutCount;
    const VkDescriptorSetLayout*    pSetLayouts;
    uint32_t                        pushConstantRangeCount;
    const VkPushConstantRange*      pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

The *sType* member of the VkPipelineLayoutCreateInfo structure should be set to VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO. The *pNext* member is reserved for use by extensions and should be set to **NULL**.

The *setLayoutCount* member specifies the number of descriptor sets to include in the layout and *pSetLayouts* is a pointer to an array of *setLayoutCount* VkDescriptorSetLayout objects describing the sets, each created with vkCreateDescriptorSetLayout.

Valid Usage

- *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

1.63.5 See Also

vkCreateDescriptorSetLayout

1.64 vkCreateQueryPool(3)

1.64.1 Name

vkCreateQueryPool - Create a new query pool object.

1.64.2 C Specification

```
VkResult vkCreateQueryPool(
    VkDevice                                    device,
    const VkQueryPoolCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkQueryPool*                                pQueryPool);
```

1.64.3 Parameters

device
The device with which to create the query pool object.

pCreateInfo
A pointer to a structure containing information to be placed in the object.

pQueryPool
A pointer to a variable which will receive the handle to the new object.

1.64.4 Description

vkCreateQueryPool creates a new query pool object using the information contained in *pCreateInfo* and the device specified in *device*. Upon success, a handle to the new query pool object is deposited in the variable pointed to by *pQueryPool*. *pCreateInfo* should point to an instance of the VkQueryPoolCreateInfo structure, the definition of which is:

```
typedef struct VkQueryPoolCreateInfo {
    VkStructureType                  sType;
    const void*                      pNext;
    VkQueryPoolCreateFlags           flags;
    VkQueryType                      queryType;
    uint32_t                         queryCount;
    VkQueryPipelineStatisticFlags    pipelineStatistics;
} VkQueryPoolCreateInfo;
```

Valid Usage

- *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

1.64.5 Return Value

Upon success, **vkCreateQueryPool** returns VK_SUCCESS and deposits the resulting query pool handle in the variable pointed to by *pQueryPool*. Upon failure, a descriptive error code is returned.

1.64.6 See Also

vkCmdResetQueryPool, vkCmdBeginQuery, vkCmdEndQuery, vkDestroyQueryPool, vkGetQueryPoolResults, vkCmdCopyQueryPoolResults

1.65 vkCreateRenderPass(3)

1.65.1 Name

vkCreateRenderPass - Create a new render pass object.

1.65.2 C Specification

```
VkResult vkCreateRenderPass(
    VkDevice                                    device,
    const VkRenderPassCreateInfo*               pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkRenderPass*                               pRenderPass);
```

1.65.3 Parameters

device
The device with which to create the render pass object.

pCreateInfo
A pointer to a structure containing information to be placed in the object.

pRenderPass
A pointer to a variable which will receive the handle to the new object.

1.65.4 Description

vkCreateRenderPass creates a new render pass object using the information contained in *pCreateInfo* and the device specified in *device*. Upon success, a handle to the new render pass object is deposited in the variable pointed to by *pRenderPass*, *pCreateInfo* should point to an instance of the VkRenderPassCreateInfo structure, the definition of which is:

```
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;
```

A render pass is a sequence of subpasses, each of which reads from some framebuffer attachments and writes to others as color and depth/stencil. The subpasses all render to the same dimensions, and fragments for pixel (x,y,layer) in one subpass only read framebuffer contents written by earlier subpasses at that same (x,y,layer) location. It is quite common for a render pass to only contain a single subpass.

Dependencies between subpasses describe ordering restrictions between them. Without dependencies, implementations may reorder or overlap execution of two subpasses.

Attachments

The attachments used in the render pass are described by a `VkAttachmentDescription` structure, defined as:

```
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                        format;
    VkSampleCountFlagBits           samples;
    VkAttachmentLoadOp              loadOp;
    VkAttachmentStoreOp             storeOp;
    VkAttachmentLoadOp              stencilLoadOp;
    VkAttachmentStoreOp             stencilStoreOp;
    VkImageLayout                   initialLayout;
    VkImageLayout                   finalLayout;
} VkAttachmentDescription;
```

The *format* and *samples* members are respectively the format and the number of samples of the image that will be used for the attachment.

The *loadOp* defines how the contents of the attachment within the render area will be treated at the beginning of the render pass. A load op of `VK_ATTACHMENT_LOAD_OP_LOAD` means the contents within the render area will be preserved; `VK_ATTACHMENT_LOAD_OP_CLEAR` means the contents within the render area will be cleared to a uniform value; `VK_ATTACHMENT_LOAD_OP_DONT_CARE` means the application intends to overwrite all samples in the render area without reading the initial contents, so their initial contents are unimportant. If the attachment format has both depth and stencil components, *loadOp* applies only to the depth data, while *stencilLoadOp* defines how the stencil data is handled. *stencilLoadOp* is ignored for other formats.

The *storeOp* defines whether data rendered to the attachment is committed to memory at the end of the render pass. `VK_ATTACHMENT_STORE_OP_STORE` means the data is committed to memory and will be available for reading after the render pass completes. `VK_ATTACHMENT_STORE_OP_DONT_CARE` means the data is not needed after rendering, and may be discarded; the contents of the attachment will be undefined inside the render area. If the attachment format has both depth and stencil components, *storeOp* applies only to the depth data, while *stencilStoreOp* defines how the stencil data is handled. *stencilStoreOp* is ignored for other formats.

initialLayout is the layout the attachment image will be in when the render pass begins.

finalLayout is the layout the attachment image will be transitioned to when the render pass ends.

Subpasses

Subpasses of a render pass are described by a `VkSubpassDescription` structure, 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;
```

The *pipelineBindPoint* indicates whether this is a compute or graphics subpass. Only graphics subpasses are currently allowed.

The *flags* member is currently unused and must be zero.

pInputAttachments lists which of the render pass's attachments will be read in the shader in the subpass, and what layout the attachment images should be transitioned to before the subpass. *inputAttachmentCount* indicates the number of input attachments. Input attachments must also be bound to the pipeline with a descriptor set.

pColorAttachments lists which of the render pass's attachments will be used as color attachments in the subpass, and what layout the attachment images should be transitioned to before the subpass. *colorAttachmentCount* indicates the number of color attachments.

Each entry in *pResolveAttachments* corresponds to an entry in *pColorAttachments*; either *pResolveAttachments* must be NULL or it must have *colorAttachmentCount* entries. If *pResolveAttachments* is not NULL, each of its elements corresponds to a color attachment (the element in *pColorAttachments* at the same index). At the end of each subpass, the subpass's color attachments will be resolved to the corresponding resolve attachments, unless the resolve attachment index is `VK_ATTACHMENT_UNUSED`.

The *depthStencilAttachment* indicates which attachment will be used for depth/stencil data and the layout it should be transitioned to before the subpass. If no depth/stencil attachment is used in the subpass, the attachment index must be `VK_ATTACHMENT_UNUSED`.

The *pPreserveAttachments* are the attachments that aren't used by a subpass, but whose contents must be preserved throughout the subpass. If the contents of an attachment are produced in one subpass and consumed in a later subpass, the attachment must be preserved in any subpasses on dependency chains from the producer to consumer. *preserveAttachmentCount* indicates the number of preserved attachments.

If a subpass uses an attachment as both an input attachment and either a color attachment or a depth/stencil attachment, all pipelines used in the subpass must disable writes to any components of the attachment format that are used as input.

Dependencies

Dependencies describe a pipeline barrier that must occur between two subpasses, usually because the destination subpass reads attachment contents written by the source subpass. Dependencies are described by `VkSubpassDependency` structures, defined as:

```
typedef struct VkSubpassDependency {
    uint32_t                srcSubpass;
    uint32_t                dstSubpass;
    VkPipelineStageFlags    srcStageMask;
    VkPipelineStageFlags    dstStageMask;
    VkAccessFlags           srcAccessMask;
    VkAccessFlags           dstAccessMask;
    VkDependencyFlags       dependencyFlags;
} VkSubpassDependency;
```

The *srcSubpass* and *dstSubpass* are producer and consumer subpasses, respectively. *srcSubpass* must be less than or equal to *dstSubpass*, so that the order of subpass descriptions is always a valid execution ordering, and so the dependency graph cannot have cycles.

The *srcStageMask*, *dstStageMask*, *outputMask*, *inputMask*, and *byRegion* describe the barrier, and have the same meaning as the VkCmdPipelineBarrier parameters and `VkMemoryBarrier` members with the same names.

If *byRegion* is `VK_TRUE`, it describes a per-region (x,y,layer) dependency, that is for each region, the *srcStageMask* stages must have finished in *srcSubpass* before any *dstStageMask* stage starts in *dstSubpass* for the same region. If *byRegion* is `VK_FALSE`, it describes a global dependency, that is the *srcStageMask* stages must have finished for all regions in *srcSubpass* before any *dstStageMask* stage starts in *dstSubpass* for any region.

Valid Usage

- *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

1.65.5 See Also

vkCmdBeginRenderPass, vkCmdEndRenderPass

1.66 vkCreateSampler(3)

1.66.1 Name

vkCreateSampler - Create a new sampler object

1.66.2 C Specification

```
VkResult vkCreateSampler(
    VkDevice                                    device,
    const VkSamplerCreateInfo*                  pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSampler*                                  pSampler);
```

1.66.3 Parameters

device
The device used to create the sampler object.

pCreateInfo
A pointer to a structure containing the parameters used to construct the sampler.

pSampler
A pointer to a variable which will receive the handle to the new sampler object.

1.66.4 Description

vkCreateSampler creates a new sampler object using the device specified in *device* and places the resulting handle in the variable whose address is given by *pSampler*. *pCreateInfo* is an instance of the VkSamplerCreateInfo structure whose definition is:

```
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;
```

The resulting sampler object should be destroyed with a call to vkDestroySampler.

Valid Usage

- `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`

1.66.5 See Also

`vkDestroySampler`, `vkCreateImage`

1.67 vkCreateSemaphore(3)

1.67.1 Name

vkCreateSemaphore - Create a new queue semaphore object.

1.67.2 C Specification

```
VkResult vkCreateSemaphore(
    VkDevice                                    device,
    const VkSemaphoreCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSemaphore*                                pSemaphore);
```

1.67.3 Parameters

device
The device with which to create the queue semaphore object.

pCreateInfo
A pointer to a structure containing information to be placed in the object.

pSemaphore
A pointer to a variable which will receive the handle to the new object.

1.67.4 Description

vkCreateSemaphore creates a new queue semaphore object using the information contained in *pCreateInfo* and the device specified in *device*. Upon success, a handle to the new queue semaphore object is deposited in the variable pointed to by *pSemaphore*. *pCreateInfo* should point to an instance of the VkSemaphoreCreateInfo structure, the definition of which is:

```
typedef struct VkSemaphoreCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkSemaphoreCreateFlags    flags;
} VkSemaphoreCreateInfo;
```

The *sType* member of the VkSemaphoreCreateInfo structure should be set to VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO. The *pNext* member is reserved for use by extensions and should be set to **NULL**.

The *flags* member of the VkSemaphoreCreateInfo structure pointed to by *pCreateInfo* contains flags defining the initial state and behavior of the semaphore. Currently, no flags are defined.

The semaphore is created in the unsignaled state and may be signaled by submitting it to a queue through a call to vkQueueSubmit.

Valid Usage

- *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

1.67.5 See Also

vkDestroySemaphore, vkQueueSubmit

1.68 vkCreateShaderModule(3)

1.68.1 Name

vkCreateShaderModule - Creates a new shader module object.

1.68.2 C Specification

```
VkResult vkCreateShaderModule(
    VkDevice                                    device,
    const VkShaderModuleCreateInfo*             pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkShaderModule*                             pShaderModule);
```

1.68.3 Parameters

device
Logical device to own the new object.

pCreateInfo
A pointer to a structure defining the shader module object to be created.

pShaderModule
Pointer to the variable to receive a handle to the new object.

1.68.4 Description

vkCreateShaderModule creates a new shader module from shader source provided by the caller. *device* is a handle to the device that is to be used to create the shader module. *pCreateInfo* is a pointer to an instance of the VkShaderModuleCreateInfo structure which contains information needed to construct the module. The definition of VkShaderModuleCreateInfo is:

```
typedef struct VkShaderModuleCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkShaderModuleCreateFlags    flags;
    size_t                       codeSize;
    const uint32_t*              pCode;
} VkShaderModuleCreateInfo;
```

The *pCode* member of *pCreateInfo* contains a pointer to an opaque code structure describing the content of the shader module. The *codeSize* member specifies the length of the data pointed to by *pCreateInfo* in bytes. The *flags* member of *pCreateInfo* is used to further control construction of the shader module. However, no flags are currently defined, *flags* is therefore reserved and should be set to zero.

Upon success, a handle to the newly created shader module object is placed in the variable that is pointed to by *pShaderModule*.

Valid Usage

- *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

1.68.5 See Also

vkDestroyShaderModule

1.69 vkDestroyBuffer(3)

1.69.1 Name

vkDestroyBuffer - Destroy a buffer object

1.69.2 C Specification

```
void vkDestroyBuffer(
    VkDevice                                    device,
    VkBuffer                                    buffer,
    const VkAllocationCallbacks*                pAllocator);
```

1.69.3 Parameters

device
: Logical device which owns the object.

buffer
: The handle of the buffer object to destroy.

1.69.4 Description

vkDestroyBuffer destroys the buffer object whose handle is specified in *buffer*. *buffer* must be a valid handle to buffer object created through a successful call to vkCreateBuffer on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a buffer object has been destroyed, its handle becomes invalid and must not be accessed again. Furthermore, any views of the buffer previously created through calls to vkCreateBufferView on the specified buffer also become invalid and should be destroyed before the parent buffer.

Valid Usage

- *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*
- Each of *device* and *buffer* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *buffer* must be externally synchronized

1.69.5 See Also

vkCreateBuffer

1.70 vkDestroyBufferView(3)

1.70.1 Name

vkDestroyBufferView - Destroy a buffer view object

1.70.2 C Specification

```
void vkDestroyBufferView(
    VkDevice                                    device,
    VkBufferView                                bufferView,
    const VkAllocationCallbacks*                pAllocator);
```

1.70.3 Parameters

device
: Logical device which owns the object.

bufferView
: The handle of the buffer view object to destroy.

1.70.4 Description

vkDestroyBufferView destroys the buffer view object whose handle is specified in *bufferView*. *bufferView* must be a valid handle to buffer view object created through a successful call to vkCreateBufferView on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a buffer view object has been destroyed, its handle becomes invalid and must not be accessed again.

Valid Usage

- *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*
- Each of *device* and *bufferView* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *bufferView* must be externally synchronized

1.70.5 See Also

vkCreateBufferView

1.71 vkDestroyCommandPool(3)

1.71.1 Name

vkDestroyCommandPool - Destroy a command pool object

1.71.2 C Specification

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

1.71.3 Parameters

device
Logical device which owns the object.

commandPool
The command pool to destroy.

1.71.4 Description

vkDestroyCommandPool destroys the command pool object whose handle is specified in *commandPool*. *commandPool* must be a valid handle to command object created through a successful call to vkCreateCommandPool on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a command pool object has been destroyed, its handle becomes invalid and must not be accessed again. Any command buffers allocated from the pool also become invalid and must not be accessed.

All command buffers allocated from the pool must be freed by a call to vkFreeCommandBuffers before the pool is destroyed. Failure to return command buffers to their command pools before destroying the pool object may result in resource leaks.

Valid Usage

- *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*
- Each of *device* and *commandPool* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All VkCommandBuffer objects allocated from *commandPool* must not be pending execution
- 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

Host Synchronization

- Host access to *commandPool* must be externally synchronized

1.71.5 See Also

vkCreateCommandPool, vkResetCommandPool

1.72 vkDestroyDescriptorPool(3)

1.72.1 Name

vkDestroyDescriptorPool - Destroy a descriptor pool object

1.72.2 C Specification

```
void vkDestroyDescriptorPool(
    VkDevice                                    device,
    VkDescriptorPool                            descriptorPool,
    const VkAllocationCallbacks*                pAllocator);
```

1.72.3 Parameters

device
Logical device which owns the object.

descriptorPool
The handle of the descriptor pool to destroy.

1.72.4 Description

vkDestroyDescriptorPool destroys the descriptor pool object whose handle is specified in *descriptorPool*. *descriptorPool* must be a valid handle to descriptor pool object created through a successful call to vkCreateDescriptorPool on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a descriptor pool object has been destroyed, its handle becomes invalid and must not be accessed again.

Any descriptor sets allocated from the pool should be freed before the pool is destroyed. Not returning descriptor sets to the pool before destroying the pool may cause a resource leak.

Valid Usage

- *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*
- Each of *device* and *descriptorPool* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *descriptorPool* must be externally synchronized

1.72.5 See Also

vkCreateDescriptorPool

1.73 vkDestroyDescriptorSetLayout(3)

1.73.1 Name

vkDestroyDescriptorSetLayout - Destroy a descriptor set layout object

1.73.2 C Specification

```
void vkDestroyDescriptorSetLayout(
    VkDevice                                    device,
    VkDescriptorSetLayout                       descriptorSetLayout,
    const VkAllocationCallbacks*                pAllocator);
```

1.73.3 Parameters

device
: Logical device which owns the object.

descriptorSetLayout
: The handle of the object to destroy.

1.73.4 Description

vkDestroyDescriptorSetLayout destroys the descriptor set layout object whose handle is specified in *descriptorSetLayout*. *descriptorSetLayout* must be a valid handle to fence object created through a successful call to vkCreateDescriptorSetLayout on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a descriptor set object has been destroyed, its handle becomes invalid and must not be accessed again.

Valid Usage

- *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*
- Each of *device* and *descriptorSetLayout* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *descriptorSetLayout* must be externally synchronized

1.73.5 See Also

vkCreateDescriptorSetLayout

1.74 vkDestroyDevice(3)

1.74.1 Name

vkDestroyDevice - Destroy a logical device.

1.74.2 C Specification

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

1.74.3 Parameters

device
A handle to the logical device to destroy.

1.74.4 Description

vkDestroyDevice destroys a logical device. It does not destroy any resources created by or associated with the device. If those resources are not destroyed, they may be leaked. Therefore, applications should ensure that all objects created through the logical device have been destroyed before destroying the device itself. Attempts to destroy **VK_NULL_HANDLE** are silently ignored.

Valid Usage

- 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
- 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

Host Synchronization

- Host access to *device* must be externally synchronized

1.74.5 See Also

vkCreateDevice, vkDestroyDevice

1.75 vkDestroyEvent(3)

1.75.1 Name

vkDestroyEvent - Destroy an event object

1.75.2 C Specification

```
void vkDestroyEvent(
    VkDevice                                    device,
    VkEvent                                     event,
    const VkAllocationCallbacks*                pAllocator);
```

1.75.3 Parameters

device
Logical device which owns the object.

event
The handle of the object to destroy.

1.75.4 Description

vkDestroyEvent destroys the event object whose handle is specified in *event*. *event* must be a valid handle to event object created through a successful call to vkCreateEvent on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a event object has been destroyed, its handle becomes invalid and must not be accessed again. Results are undefined if a command buffer is waiting on a event when the event is destroyed.

Valid Usage

- *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*
- Each of *device* and *event* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *event* must be externally synchronized

1.75.5 See Also

vkCreateEvent

1.76 vkDestroyFence(3)

1.76.1 Name

vkDestroyFence - Destroy a fence object

1.76.2 C Specification

```
void vkDestroyFence(
    VkDevice                                    device,
    VkFence                                     fence,
    const VkAllocationCallbacks*                pAllocator);
```

1.76.3 Parameters

device
Logical device which owns the object.

fence
The handle of the object to destroy.

1.76.4 Description

`vkDestroyFence` destroys the fence object whose handle is specified in *`fence`*. *`fence`* must be a valid handle to fence object created through a successful call to `vkCreateFence` on *`device`*. Attempts to destroy **`VK_NULL_HANDLE`** are silently ignored. After a fence object has been destroyed, its handle becomes invalid and must not be accessed again. Results are undefined if another thread is waiting on a fence when the fence is destroyed.

Valid Usage

- *`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`*
- Each of *`device`* and *`fence`* that are valid handles must have been created, allocated or retrieved from the same `VkPhysicalDevice`
- *`fence`* must not be associated with any queue command that has not yet completed execution on that queue
- 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`

Host Synchronization

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

1.76.5 See Also

`vkCreateFence`

1.77 vkDestroyFramebuffer(3)

1.77.1 Name

vkDestroyFramebuffer - Destroy a framebuffer object

1.77.2 C Specification

```
void vkDestroyFramebuffer(
    VkDevice                                    device,
    VkFramebuffer                               framebuffer,
    const VkAllocationCallbacks*                pAllocator);
```

1.77.3 Parameters

device
: Logical device which owns the object.

framebuffer
: The handle of the object to destroy.

1.77.4 Description

vkDestroyFramebuffer destroys the framebuffer object whose handle is specified in *framebuffer*. *framebuffer* must be a valid handle to framebuffer object created through a successful call to vkCreateFramebuffer on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a framebuffer object has been destroyed, its handle becomes invalid and must not be accessed again.

Valid Usage

- *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*
- Each of *device* and *framebuffer* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *framebuffer* must be externally synchronized

1.77.5 See Also

vkCreateFramebuffer

1.78 vkDestroyImage(3)

1.78.1 Name

vkDestroyImage - Destroy an image object

1.78.2 C Specification

```
void vkDestroyImage(
    VkDevice                                    device,
    VkImage                                     image,
    const VkAllocationCallbacks*                pAllocator);
```

1.78.3 Parameters

device
Logical device which owns the object.

image
The handle of the object to destroy.

1.78.4 Description

vkDestroyImage destroys the image object whose handle is specified in *image*. *image* must be a valid handle to an image object created through a successful call to vkCreateImage on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After an image has been destroyed, its handle becomes invalid and must not be accessed again. Additionally, views of images immediately become invalid once the parent image has been destroyed.

Valid Usage

- *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*
- Each of *device* and *image* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *image* must be externally synchronized

1.78.5 See Also

vkCreateImage

1.79 vkDestroyImageView(3)

1.79.1 Name

vkDestroyImageView - Destroy an image view object

1.79.2 C Specification

```
void vkDestroyImageView(
    VkDevice                                    device,
    VkImageView                                 imageView,
    const VkAllocationCallbacks*                pAllocator);
```

1.79.3 Parameters

device
Logical device which owns the object.

imageView
The handle of the image view object to destroy.

1.79.4 Description

vkDestroyImageView destroys the image view object whose handle is specified in *imageView*. *imageView* must be a valid handle to an image view object created through a successful call to vkCreateImageView on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After an image view has been destroyed, its handle becomes invalid and must not be accessed again.

Valid Usage

- *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*
- Each of *device* and *imageView* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *imageView* must be externally synchronized

1.79.5 See Also

vkCreateImage, vkDestroyImage, vkCreateImageView

1.80 vkDestroyInstance(3)

1.80.1 Name

vkDestroyInstance - Destroy an instance of Vulkan.

1.80.2 C Specification

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

1.80.3 Parameters

instance
Vulkan instance to release.

1.80.4 Description

vkDestroyInstance destroys an instance of Vulkan. After destruction of the instance, all devices (logical and physical) and any objects created by those devices become in valid and should not be accessed. However, objects allocated directly or indirectly through the instance are not destroyed automatically and so may be leaked. Applications should destroy all objects created through *instance* before destroying the instance itself.

Valid Usage

- 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
- 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

Host Synchronization

- Host access to *instance* must be externally synchronized

1.80.5 See Also

vkCreateInstance, vkCreateDevice, vkDestroyDevice

1.81 vkDestroyPipeline(3)

1.81.1 Name

vkDestroyPipeline - Destroy a pipeline object

1.81.2 C Specification

```
void vkDestroyPipeline(
    VkDevice                                    device,
    VkPipeline                                  pipeline,
    const VkAllocationCallbacks*                pAllocator);
```

1.81.3 Parameters

device
Logical device which owns the object.

pipeline
The handle of the object to destroy.

1.81.4 Description

vkDestroyPipeline destroys the pipeline object whose handle is specified in *pipeline*. *pipeline* must be a valid handle to a pipeline created through a successful call to vkCreateGraphicsPipelines or vkCreateComputePipelines on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. Once a pipeline has been destroyed, its handle becomes invalid and must not be accessed again.

Valid Usage

- *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*
- Each of *device* and *pipeline* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *pipeline* must be externally synchronized

1.81.5 See Also

vkCreateGraphicsPipelines, vkCreateComputePipelines

1.82 vkDestroyPipelineCache(3)

1.82.1 Name

vkDestroyPipelineCache - Destroy a pipeline cache object

1.82.2 C Specification

```
void vkDestroyPipelineCache(
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    const VkAllocationCallbacks*                pAllocator);
```

1.82.3 Parameters

device
: Logical device which owns the object.

pipelineCache
: The handle of the object to destroy.

1.82.4 Description

vkDestroyPipelineCache destroys the pipeline cache object whose handle is specified in *pipelineCache*, which must be a valid handle to a pipeline cache object that was created by a successful call to vkCreatePipelineCache on *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a pipeline cache has been destroyed, its handle becomes invalid and should not be accessed again.

Valid Usage

- *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*
- Each of *device* and *pipelineCache* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *pipelineCache* must be externally synchronized

1.82.5 See Also

vkCreatePipelineCache, vkGetPipelineCacheData, vkMergePipelineCaches

1.83 vkDestroyPipelineLayout(3)

1.83.1 Name

vkDestroyPipelineLayout - Destroy a pipeline layout object

1.83.2 C Specification

```
void vkDestroyPipelineLayout(
    VkDevice                                    device,
    VkPipelineLayout                            pipelineLayout,
    const VkAllocationCallbacks*                pAllocator);
```

1.83.3 Parameters

device
Logical device which owns the object.

pipelineLayout
The handle of the object to destroy.

1.83.4 Description

vkDestroyPipelineLayout destroys the pipeline layout object whose handle is specified in *pipelineLayout*, which must be a valid handle to a pipeline layout that was created by *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a pipeline layout has been destroyed, its handle becomes invalid and should not be accessed again.

Valid Usage

- *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*
- Each of *device* and *pipelineLayout* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *pipelineLayout* must be externally synchronized

1.83.5 See Also

vkCreatePipelineLayout

1.84 vkDestroyQueryPool(3)

1.84.1 Name

vkDestroyQueryPool - Destroy a query pool object

1.84.2 C Specification

```
void vkDestroyQueryPool(
    VkDevice                                    device,
    VkQueryPool                                 queryPool,
    const VkAllocationCallbacks*                pAllocator);
```

1.84.3 Parameters

device
Logical device which owns the object.

queryPool
The handle of the object to destroy.

1.84.4 Description

vkDestroyQueryPool destroys the query pool whose handle is specified in *queryPool*, which must be a valid handle to a query pool that was created by *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a query pool has been destroyed, its handle becomes invalid and should not be accessed again.

Valid Usage

- *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*
- Each of *device* and *queryPool* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *queryPool* must be externally synchronized

1.84.5 See Also

vkCreateQueryPool, vkCmdResetQueryPool, vkCmdBeginQuery, vkCmdEndQuery, vkCmdCopyQueryPoolResults, vkGetQueryPoolResults

1.85 vkDestroyRenderPass(3)

1.85.1 Name

vkDestroyRenderPass - Destroy a render pass object

1.85.2 C Specification

```
void vkDestroyRenderPass(
    VkDevice                                    device,
    VkRenderPass                                renderPass,
    const VkAllocationCallbacks*                pAllocator);
```

1.85.3 Parameters

device
Logical device which owns the object.

renderPass
The handle of the object to destroy.

1.85.4 Description

vkDestroyRenderPass destroys the render pass whose handle is specified in *renderPass*, which must be a valid handle to a render pass that was created by *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a render pass has been destroyed, its handle becomes invalid and should not be accessed again.

Valid Usage

- *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*
- Each of *device* and *renderPass* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *renderPass* must be externally synchronized

1.85.5 See Also

vkCreateRenderPass

1.86 vkDestroySampler(3)

1.86.1 Name

vkDestroySampler - Destroy a sampler object

1.86.2 C Specification

```
void vkDestroySampler(
    VkDevice                                    device,
    VkSampler                                   sampler,
    const VkAllocationCallbacks*                pAllocator);
```

1.86.3 Parameters

device
Logical device which owns the object.

sampler
The handle of the object to destroy.

1.86.4 Description

vkDestroySampler destroys the sampler whose handle is specified in *sampler*, which must be a valid handle to a sampler that was created by *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. After a sampler has been destroyed, its handle becomes invalid and should not be accessed again.

Valid Usage

- *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*
- Each of *device* and *sampler* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *sampler* must be externally synchronized

1.86.5 See Also

vkCreateSampler

1.87 vkDestroySemaphore(3)

1.87.1 Name

vkDestroySemaphore - Destroy a semaphore object

1.87.2 C Specification

```
void vkDestroySemaphore(
    VkDevice                                    device,
    VkSemaphore                                 semaphore,
    const VkAllocationCallbacks*                pAllocator);
```

1.87.3 Parameters

device
Logical device which owns the object.

semaphore
The handle of the object to destroy.

1.87.4 Description

vkDestroySemaphore destroys the semaphore whose handle is specified in *semaphore*, which must be a valid handle to a semaphore that was created by device *device*. Attempts to destroy **VK_NULL_HANDLE** are silently ignored. Once a semaphore has been destroyed, its handle becomes invalid and must not be reused.

Valid Usage

- *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*
- Each of *device* and *semaphore* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- *semaphore* must not be associated with any queue command that has not yet completed execution on that queue
- 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

Host Synchronization

- Host access to *semaphore* must be externally synchronized

1.87.5 See Also

vkCreateSemaphore

1.88 vkDestroyShaderModule(3)

1.88.1 Name

vkDestroyShaderModule - Destroy a shader module module

1.88.2 C Specification

```
void vkDestroyShaderModule(
    VkDevice                                    device,
    VkShaderModule                              shaderModule,
    const VkAllocationCallbacks*                pAllocator);
```

1.88.3 Parameters

device
Logical device which owns the object.

shaderModule
The handle of the object to destroy.

1.88.4 Description

vkDestroyShaderModule destroys the shader module specified in *shaderModule*, which must be a valid handle to a shader module owned by *device*. An attempt to destroy the **VK_NULL_HANDLE** handle are silently ignored. After the shader module has been destroyed its handle becomes invalid and it should not be accessed again.

Valid Usage

- *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*
- Each of *device* and *shaderModule* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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

Host Synchronization

- Host access to *shaderModule* must be externally synchronized

1.88.5 See Also

vkCreateShaderModule, vkCreateShader, vkDestroyShader

1.89 vkDeviceWaitIdle(3)

1.89.1 Name

vkDeviceWaitIdle - Wait for a device to become idle.

1.89.2 C Specification

```
VkResult vkDeviceWaitIdle(
    VkDevice                                    device);
```

1.89.3 Parameters

device
The handle to the device to idle.

1.89.4 Description

vkDeviceWaitIdle waits for the device specified by *device* to complete all work submitted by the application and become idle. It is logically equivalent to calling vkQueueWaitIdle on all queues associated with the device.

Valid Usage

- *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

1.89.5 See Also

vkQueueWaitIdle

1.90 vkEndCommandBuffer(3)

1.90.1 Name

vkEndCommandBuffer - Finish recording a command buffer

1.90.2 C Specification

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

1.90.3 Parameters

commandBuffer
A handle to the command buffer for which recording is to end.

1.90.4 Description

vkEndCommandBuffer ends the recording of a command buffer. The command buffer must be in the recording state. After recording of a primary command buffer is completed, it may be submitted to a queue using vkQueueSubmit. After recording of a secondary command buffer is completed, it may be called from a primary command buffer by a call to vkCmdExecuteCommands. No further modification to a completed command buffer may be performed until vkResetCommandBuffer is called on it.

Valid Usage

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

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

1.90.5 See Also

vkAllocateCommandBuffers, vkFreeCommandBuffers, vkBeginCommandBuffer, vkResetCommandBuffer, vkCmdExecuteCommands

1.91 vkEnumerateDeviceExtensionProperties(3)

1.91.1 Name

vkEnumerateDeviceExtensionProperties - Returns properties of available physical device extensions.

1.91.2 C Specification

```
VkResult vkEnumerateDeviceExtensionProperties(
    VkPhysicalDevice                            physicalDevice,
    const char*                                 pLayerName,
    uint32_t*                                   pPropertyCount,
    VkExtensionProperties*                      pProperties);
```

1.91.3 Parameters

physicalDevice
Physical device to query.

pLayerName
Optional layer name to query.

pPropertyCount
Count indicating number of `VkExtensionProperties` pointed to by *pProperties*.

pProperties
Pointer to an array of `VkExtensionProperties`.

1.91.4 Description

vkEnumerateDeviceExtensionProperties retrieves properties for extensions on a physical device whose handle is given in *physicalDevice*. To determine the extensions implemented by a layer set *pLayerName* to point to the layer's name and any returned extensions are implemented by that layer. Setting *pLayerName* to NULL will return the available non-layer extensions. *pPropertyCount* must be set to the size of the VkExtensionProperties array pointed to by *pProperties*. The *pProperties* should point to an array of VkExtensionProperties to be filled out or null. If null, **vkEnumerateDeviceExtensionProperties** will update pPropertyCount with the number of extensions found. The definition of `VkExtensionProperties` is as follows:

```
typedef struct VkExtensionProperties {
    char        extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t    specVersion;
} VkExtensionProperties;
```

Valid Usage

- *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
- If *pLayerName* is not `NULL`, it must be the name of a layer returned by `vkEnumerateDeviceLayerProperties`

Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

1.91.5 See Also

vkEnumerateDeviceLayerProperties, vkCreateDevice

1.92 vkEnumerateDeviceLayerProperties(3)

1.92.1 Name

vkEnumerateDeviceLayerProperties - Returns properties of available physical device layers.

1.92.2 C Specification

```
VkResult vkEnumerateDeviceLayerProperties(
    VkPhysicalDevice                            physicalDevice,
    uint32_t*                                   pPropertyCount,
    VkLayerProperties*                          pProperties);
```

1.92.3 Parameters

physicalDevice
Physical device to query.

pPropertyCount
Count indicating number of `VkLayerProperties` pointed to by *pProperties*.

pProperties
Pointer to an array of `VkLayerProperties`.

1.92.4 Description

vkEnumerateDeviceLayerProperties retrieves properties for layers on a physical device whose handle is given in *physicalDevice*. *pPropertyCount* must be a valid pointer to an integer set to the size of the `VkLayerProperties` array pointed to by *pProperties*. *pProperties* must be **NULL** or a pointer to an array of `VkLayerProperties` to be filled out. If **NULL**, **vkEnumerateDeviceLayerProperties** will update *pPropertyCount* with the number of layers found. The definition of `VkLayerProperties` is as follows:

```
typedef struct VkLayerProperties {
    char        layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t    specVersion;
    uint32_t    implementationVersion;
    char        description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

Valid Usage

- *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

1.92.5 See Also

vkEnumerateDeviceLayerProperties, vkCreateDevice

1.93 vkEnumerateInstanceExtensionProperties(3)

1.93.1 Name

vkEnumerateInstanceExtensionProperties - Returns up to requested number of global extension properties.

1.93.2 C Specification

```
VkResult vkEnumerateInstanceExtensionProperties(
    const char*                                 pLayerName,
    uint32_t*                                   pPropertyCount,
    VkExtensionProperties*                      pProperties);
```

1.93.3 Parameters

pLayerName
Pointer to optional layer name. If not null, will only return extension properties for the requested layer.

pPropertyCount
Pointer to count indicating space available on input and structures returned on output.

pProperties
Pointer to a data structure to receive the results.

1.93.4 Description

vkEnumerateInstanceExtensionProperties retrieves properties for global extensions of the loader or the optionally specified layer. *pProperties* points to an array of `VkExtensionProperties` where the return data will be stored. If NULL, **vkEnumerateInstanceExtensionProperties** will update the count with the number of global extensions found. *pPropertyCount* must point to a count indicating the number of `VkExtensionProperties` structures available. The definition of `VkExtensionProperties` is as follows:

```
typedef struct VkExtensionProperties {
    char        extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t    specVersion;
} VkExtensionProperties;
```

Valid Usage

- 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
- If *pLayerName* is not `NULL`, it must be the name of a layer returned by `vkEnumerateInstanceLayerProperties`

Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

1.93.5 Return Value

Upon success, **vkEnumerateInstanceExtensionProperties** returns VK_SUCCESS or VK_INCOMPLETE. VK_INCOMPLETE indicates that the number of extension properties found exceeds the given count. An app will need to call again with a larger array and count to get all available extension properties. The number of available extensions could change from one call to the next if an application updates or installs Vulkan components.

1.93.6 See Also

vkEnumerateInstanceLayerProperties, vkCreateInstance

1.94 vkEnumerateInstanceLayerProperties(3)

1.94.1 Name

vkEnumerateInstanceLayerProperties - Returns up to requested number of global layer properties.

1.94.2 C Specification

```
VkResult vkEnumerateInstanceLayerProperties(
    uint32_t*                                   pPropertyCount,
    VkLayerProperties*                          pProperties);
```

1.94.3 Parameters

pPropertyCount
Pointer to count indicating space available on input and structures returned on output.

pProperties
Pointer to a array to receive the results.

1.94.4 Description

vkEnumerateInstanceLayerProperties retrieves properties for global layers. *pPropertyCount* must be a valid pointer to an integer set to the size of the VkLayerProperties array pointed to by *pProperties*. *pProperties* must be **NULL** or a pointer to an array of VkLayerProperties to be filled out. If **NULL**, **vkEnumerateInstanceLayerProperties** will update *pPropertyCount* with the number of layers found. The definition of VkLayerProperties is as follows:

```
typedef struct VkLayerProperties {
    char        layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t    specVersion;
    uint32_t    implementationVersion;
    char        description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

Valid Usage

- *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

1.94.5 See Also

vkEnumerateInstanceExtensionProperties, vkCreateInstance

1.95 vkEnumeratePhysicalDevices(3)

1.95.1 Name

vkEnumeratePhysicalDevices - Enumerates the physical devices accessible to a Vulkan instance.

1.95.2 C Specification

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

1.95.3 Parameters

instance
A handle to the instance to be used to enumerate devices.

pPhysicalDeviceCount
A pointer to a variable containing the maximum number of devices to enumerate.

pPhysicalDevices
A pointer to an array that will be filled with handles to the enumerated devices.

1.95.4 Description

vkEnumeratePhysicalDevices generates a list of the physical devices accessible to the instance of Vulkan specified in *instance*.

pPhysicalDeviceCount is a pointer to a variable which contains the number of devices to enumerate. *pPhysicalDeviceCount* must not be **NULL**. *pPhysicalDevices* is a pointer to an array of VkPhysicalDevice handles which will be filled with handles to the enumerated devices.

If *pPhysicalDevices* is **NULL**, then the initial value of the variable pointed to by *pPhysicalDeviceCount* is ignored and this variable is overwritten with the number of physical devices accessible to *instance*.

If *pPhysicalDevices* is not **NULL**, then *pPhysicalDeviceCount* should point to a variable that has been initialized with the size of the array pointed to by *pPhysicalDevices*. No more than this number of physical device handles will be written into the output array. The actual number of device handles written into *pPhysicalDevices* is then written into the variable pointed to by *pPhysicalDeviceCount*.

Valid Usage

- *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

1.95.5 See Also

vkGetPhysicalDeviceFeatures, vkCreateDevice

1.96 vkFlushMappedMemoryRanges(3)

1.96.1 Name

vkFlushMappedMemoryRanges - Flush mapped memory ranges.

1.96.2 C Specification

```
VkResult vkFlushMappedMemoryRanges(
    VkDevice                                    device,
    uint32_t                                    memoryRangeCount,
    const VkMappedMemoryRange*                  pMemoryRanges);
```

1.96.3 Parameters

device
Logical device which owns the specified memory ranges.

memoryRangeCount
Number of memory ranges described by *pMemoryRanges*.

pMemoryRanges
Mapped memory ranges to flush.

1.96.4 Description

vkFlushMappedMemoryRanges flushes zero more more ranges of a mapped memory objects. *device* is a handle to the device that owns the memory objects to be flushed. *memoryRangeCount* is the number of ranges to flush and *pMemoryRanges* points to an array of *memoryRangeCount* instances of the VkMappedMemoryRange structure, each defining a region of memory to flush. The definition of VkMappedMemoryRange is:

```
typedef struct VkMappedMemoryRange {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceMemory     memory;
    VkDeviceSize       offset;
    VkDeviceSize       size;
} VkMappedMemoryRange;
```

For each element of the *pMemoryRanges* array, *memory* is the memory object containing the mapped range, *offset* is the location of the start of the range within *memory*, and *size* is the size of the region to flush. Both *offset* and *size* are specified in bytes.

If any referenced region of the memory object is not mapped or extends beyond the bounds of the memory object then the command has no effect on that region, but is still honored for other regions in the array. Multiple regions inside the same memory object may be contained in *pMemoryRanges*, including ranges that overlap one another.

Flushing memory ranges ensures that any writes performed by the host become visible to commands subsequently executing on devices with references to that memory. **vkFlushMappedMemoryRanges** has no effect with respect to writes performed by the device.

Valid Usage

- *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

1.96.5 See Also

vkMapMemory, vkUnmapMemory, vkAllocateMemory, vkFreeMemory

1.97 vkFreeCommandBuffers(3)

1.97.1 Name

vkFreeCommandBuffers - Free command buffers.

1.97.2 C Specification

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

1.97.3 Parameters

device
A handle to the device that owns the command pool and command buffers referenced by the command.

commandPool
A handle to the command pool which owns the command buffers.

commandBufferCount
The number of command buffers to free.

pCommandBuffers
A pointer to an array of handles to the command buffers to free.

1.97.4 Description

vkFreeCommandBuffers frees *commandBufferCount* command buffers, returning their resources to the pool specified in *commandPool*. *pCommandBuffers* is a pointer to an array of *commandBufferCount* VkCommandBuffer handles to the command buffers to free. Each command buffer in the array must have been allocated from the pool specified in *commandPool* through a call to vkAllocateCommandBuffers. *device* must be a handle to the device that owns both *commandPool* and all of the command buffers referenced from the array pointed to by *pCommandBuffers*.

After command buffers are freed, they may not be referenced again. A command buffer must not be freed while it is in flight.

Valid Usage

- *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*
- Each of *device*, *commandPool* and the elements of *pCommandBuffers* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All elements of *pCommandBuffers* must not be pending execution
- *pCommandBuffers* must be a pointer to an array of *commandBufferCount* VkCommandBuffer handles, each element of which must either be a valid handle or VK_NULL_HANDLE

Host Synchronization

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

1.97.5 See Also

vkAllocateCommandBuffers, vkResetCommandBuffer, vkBeginCommandBuffer, vkEndCommandBuffer, vkQueueSubmit

1.98 vkFreeDescriptorSets(3)

1.98.1 Name

vkFreeDescriptorSets - Free one or more descriptor sets

1.98.2 C Specification

```
VkResult vkFreeDescriptorSets(
    VkDevice                                    device,
    VkDescriptorPool                            descriptorPool,
    uint32_t                                    descriptorSetCount,
    const VkDescriptorSet*                      pDescriptorSets);
```

1.98.3 Parameters

device
The device that owns the descriptor sets.

descriptorPool
The descriptor pool that the descriptor sets were allocated from.

descriptorSetCount
The number of descriptor sets to free.

pDescriptorSets
An array of *descriptorSetCount* variables containing the descriptor set handles to free.

1.98.4 Description

vkFreeDescriptorSets frees descriptor sets. *device* is a handle to the device that owns the descriptor pool specified in *descriptorPool*, which must be the pool from which the sets were allocated. Freeing a descriptor set returns its descriptors to the pool from which it was allocated but does not necessarily free resources associated with the set. *pDescriptorSets* is a pointer to an array of descriptor set handles returned from previous calls to vkAllocateDescriptorSets. *descriptorSetCount* specifies the number of descriptor set handles in the *pDescriptorSets* array.

To actually free resources associated with descriptor pools, call vkDestroyDescriptorPool.

Valid Usage

- *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*
- Each of *device*, *descriptorPool* and the elements of *pDescriptorSets* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- 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
- *descriptorPool* must have been created with the VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT flag

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

1.98.5 See Also

vkAllocateDescriptorSets, vkCreateDescriptorPool, vkDestroyDescriptorPool.

1.99 vkFreeMemory(3)

1.99.1 Name

vkFreeMemory - Free GPU memory

1.99.2 C Specification

```
void vkFreeMemory(
    VkDevice                                    device,
    VkDeviceMemory                              memory,
    const VkAllocationCallbacks*                pAllocator);
```

1.99.3 Parameters

device
The logical device which owns the memory object.

mem
The memory object to free.

1.99.4 Description

vkFreeMemory frees the memory object whose handle is given in *mem*. After the memory is freed, *mem* becomes invalid and should no longer be used. Further, any resource to which the memory is bound become invalid and should not be referenced. Such objects should be destroyed or rebound to a new memory object (if allowed).

Valid Usage

- *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*
- Each of *device* and *memory* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *memory* (via images or buffers) must have completed execution

Host Synchronization

- Host access to *memory* must be externally synchronized

1.99.5 See Also

vkAllocateMemory

1.100 vkGetBufferMemoryRequirements(3)

1.100.1 Name

vkGetBufferMemoryRequirements - Returns the memory requirements for specified Vulkan object.

1.100.2 C Specification

```
void vkGetBufferMemoryRequirements(
    VkDevice                                    device,
    VkBuffer                                    buffer,
    VkMemoryRequirements*                       pMemoryRequirements);
```

1.100.3 Parameters

device
: Logical device which owns *buffer*.

buffer
: Object to query.

pMemoryRequirements
: Pointer to a data structure to receive the result of the query.

1.100.4 Description

vkGetBufferMemoryRequirements retrieves memory requirements for the buffer whose handle is given in *buffer*. The *pMemoryRequirements* parameter should point to an instance of an VkMemoryRequirements structure which will be filled with the memory requirements of the buffer object. The definition of VkMemoryRequirements is as follows:

```
typedef struct VkMemoryRequirements {
    VkDeviceSize    size;
    VkDeviceSize    alignment;
    uint32_t        memoryTypeBits;
} VkMemoryRequirements;
```

The *size* member of VkMemoryRequirements reports the size of the memory allocation, measured in bytes, required by the buffer. *alignment* reports the required alignment of the memory allocation, also measured in bytes. When memory is bound to the buffer object, the offset of the range within the memory object must be an integer multiple of this value. The *memoryTypeBits* member is a bitfield with each set bit representing a valid memory type. Memory types for a device may be determined by calling vkGetPhysicalDeviceMemoryProperties. The least significant bit if *memoryTypeBits* represents the first memory type returned from vkGetPhysicalDeviceMemoryProperties, the next bit represents the second memory type and so on.

Valid Usage

- *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*
- Each of *device* and *buffer* must have been created, allocated or retrieved from the same VkPhysicalDevice

1.100.5 See Also

vkBindImageMemory

1.101 vkGetDeviceMemoryCommitment(3)

1.101.1 Name

vkGetDeviceMemoryCommitment - Query the current commitment for a VkDeviceMemory

1.101.2 C Specification

```
void vkGetDeviceMemoryCommitment(
    VkDevice                                    device,
    VkDeviceMemory                              memory,
    VkDeviceSize*                               pCommittedMemoryInBytes);
```

1.101.3 Parameters

device
The device object from which *memory* was allocated.

memory
The device memory object to query.

pCommittedMemoryInBytes
Pointer to a variable which will receive the current memory commitment, in bytes.

1.101.4 Description

vkGetDeviceMemoryCommitment queries the commitment status of a VkDeviceMemory that was created with the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT set.

The number of bytes committed for the given memory object is returned in the *pCommittedMemoryInBytes* pointer.

Valid Usage

- *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*
- Each of *device* and *memory* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *memory* must have been created with a memory type that reports VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT

1.101.5 See Also

vkGetPhysicalDeviceMemoryProperties, vkAllocateMemory, vkFreeMemory

1.102 vkGetDeviceProcAddr(3)

1.102.1 Name

vkGetDeviceProcAddr - Return a function pointer for a command

1.102.2 C Specification

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

1.102.3 Parameters

device
The VkDevice whose function pointer to query.

pName
The name of the command.

1.102.4 Description

vkGetDeviceProcAddr returns a function pointer for the command specified in *pName* as it corresponds to *device*. Depending on the operating system, supporting components, software environment and hardware topology, the function pointer returned for a single command name may be different for different values of *device*.

Device-specific function pointers only exist for commands that take a device-child object as their first parameter. In the core API these are VkDevice, VkQueue, and VkCommandBuffer, though extensions may introduce additional dispatchable device-child object types. **vkGetDeviceProcAddr** will return **NULL** when *pName* is not one of these commands.

Valid Usage

- *device* must be a valid VkDevice handle
- *pName* must be a null-terminated string
- *pName* must be the name of a supported command that has a first parameter of type VkDevice, VkQueue or VkCommandBuffer, either in the core API or an enabled extension

1.102.5 Return Value

Upon success, **vkGetDeviceProcAddr** returns a function pointer (PFN_vkVoidFunction) for the command specified in *pName*. If *pName* is not supported by the device or has no corresponding *device*, then **vkGetDeviceProcAddr** returns **NULL**.

1.102.6 See Also

vkGetInstanceProcAddr, vkCreateDevice

1.103 vkGetDeviceQueue(3)

1.103.1 Name

vkGetDeviceQueue - Get a queue handle from a device.

1.103.2 C Specification

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

1.103.3 Parameters

device
: Handle to the device that is the owner of the queue.

queueFamilyIndex
: The family index of the queue within the device.

queueIndex
: The index of the queue within the queue family.

pQueue
: A pointer to a variable that is to receive the resulting handle.

1.103.4 Description

vkGetDeviceQueue retrieves a handle to a specified queue from the device specified in *device*. The queue is identified by its family index, specified in *queueFamilyIndex* and its index within the family, specified in *queueIndex*. *pQueue* is a pointer to a variable that will receive the resulting handle.

Valid Usage

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

1.103.5 See Also

`vkGetPhysicalDeviceFeatures`, `vkGetPhysicalDeviceQueueFamilyProperties`

1.104 vkGetEventStatus(3)

1.104.1 Name

vkGetEventStatus - Retrieve the status of an event object.

1.104.2 C Specification

```
VkResult vkGetEventStatus(
    VkDevice                                    device,
    VkEvent                                     event);
```

1.104.3 Parameters

device
Logical device which owns the event.

event
A handle to the event whose status to retrieve.

1.104.4 Description

vkGetEventStatus retrieves the status of the event object specified in *event*. Event objects cannot be directly waited for by the host although it is possible to wait within a command buffer for an event to become signaled by calling vkCmdWaitEvents. Events are set from within a command buffer by calling vkCmdSetEvent and may be reset by calling vkCmdResetEvent. On the host, events may be set and reset by calling vkSetEvent and vkResetEvent, respectively.

Valid Usage

- *device* must be a valid VkDevice handle
- *event* must be a valid VkEvent handle
- *event* must have been created, allocated or retrieved from *device*
- Each of *device* and *event* must have been created, allocated or retrieved from the same VkPhysicalDevice

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

1.104.5 See Also

vkSetEvent, vkResetEvent, vkCmdSetEvent, vkCmdResetEvent, vkCmdWaitEvents

1.105 vkGetFenceStatus(3)

1.105.1 Name

vkGetFenceStatus - Return the status of a fence.

1.105.2 C Specification

```
VkResult vkGetFenceStatus(
    VkDevice                                    device,
    VkFence                                     fence);
```

1.105.3 Parameters

device
Logical device which owns *fence*.

fence
The fence whose status to return.

1.105.4 Description

vkGetFenceStatus returns the immediate status of the fence whose handle is given in *fence*. Fences are initially created in the unsignaled state and are associated with submissions to queues through a call to vkQueueSubmit. Fences are signaled by the system when work invoked by vkQueueSubmit completes. Fences may subsequently be reset by calling vkResetFences. To wait for one or more fences to become signaled, it is recommended that vkWaitForFences be used in preference to repeatedly polling vkGetFenceStatus.

Valid Usage

- *device* must be a valid VkDevice handle
- *fence* must be a valid VkFence handle
- *fence* must have been created, allocated or retrieved from *device*
- Each of *device* and *fence* must have been created, allocated or retrieved from the same VkPhysicalDevice

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

1.105.5 Return Value

Upon success, **vkGetFenceStatus** returns the status of the fence, which may be one of:

- `VK_SUCCESS` indicates that the fence has completed (its status is signaled).
- `VK_NOT_READY` indicates that the fence has not yet completed (its status is unsignaled).

Upon failure, a descriptive error code is returned.

1.105.6 See Also

`vkCreateFence`, `vkWaitForFences`, `vkQueueSubmit`

1.106 vkGetImageMemoryRequirements(3)

1.106.1 Name

vkGetImageMemoryRequirements - Returns the memory requirements for specified Vulkan object.

1.106.2 C Specification

```
void vkGetImageMemoryRequirements(
    VkDevice                                    device,
    VkImage                                     image,
    VkMemoryRequirements*                       pMemoryRequirements);
```

1.106.3 Parameters

device
Logical device which owns *image*.

image
Object to query.

pMemoryRequirements
Pointer to a data structure to receive the result of the query.

1.106.4 Description

vkGetImageMemoryRequirements retrieves memory requirements for the image object whose handle is given in *image*. The *pMemoryRequirements* parameter should point to an instance of an VkMemoryRequirements structure which will be filled with the memory requirements of the image object. The definition of VkMemoryRequirements is as follows:

```
typedef struct VkMemoryRequirements {
    VkDeviceSize    size;
    VkDeviceSize    alignment;
    uint32_t        memoryTypeBits;
} VkMemoryRequirements;
```

The *size* member of VkMemoryRequirements reports the size of the memory allocation, measured in bytes, required by the image. *alignment* reports the required alignment of the memory allocation, also measured in bytes. When memory is bound to the image object, the offset of the range within the memory object must be an integer multiple of this value. The *memoryTypeBits* member is a bitfield with each set bit representing a valid memory type. Memory types for a device may be determined by calling vkGetPhysicalDeviceMemoryProperties. The least significant bit if *memoryTypeBits* represents the first memory type returned from vkGetPhysicalDeviceMemoryProperties, the next bit represents the second memory type and so on.

Valid Usage

- *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*
- Each of *device* and *image* must have been created, allocated or retrieved from the same VkPhysicalDevice

1.106.5 See Also

vkBindImageMemory, vkGetPhysicalDeviceMemoryProperties

1.107 vkGetImageSparseMemoryRequirements(3)

1.107.1 Name

vkGetImageSparseMemoryRequirements - Query the memory requirements for a sparse image.

1.107.2 C Specification

```
void vkGetImageSparseMemoryRequirements(
    VkDevice                                    device,
    VkImage                                     image,
    uint32_t*                                   pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements*            pSparseMemoryRequirements);
```

1.107.3 Parameters

device
A handle to the device that owns the image being queried.

image
A handle to the image to be queried.

pSparseMemoryRequirementCount
On input, a pointer to a variable containing the number of elements in the array pointed to by *pSparseMemoryRequirements*. On output, this variable is overwritten with the number of elements written into *pSparseMemoryRequirements*.

pSparseMemoryRequirements
A pointer to an array of structures that will be filled with the requested information.

1.107.4 Description

vkGetImageSparseMemoryRequirements queries the device specified in *device* for the memory requirements of the sparse image specified in *image*, which must be a handle to a sparse image.

pSparseMemoryRequirementCount is a pointer to a variable which, on input to the command contains the number of elements in the array pointed to by *pSparseMemoryRequirements*. This is an array of VkSparseImageMemoryRequirements structures, the definition of which is:

```
typedef struct VkSparseImageMemoryRequirements {
    VkSparseImageFormatProperties    formatProperties;
    uint32_t                         imageMipTailFirstLod;
    VkDeviceSize                     imageMipTailSize;
    VkDeviceSize                     imageMipTailOffset;
    VkDeviceSize                     imageMipTailStride;
} VkSparseImageMemoryRequirements;
```

Within VkSparseImageMemoryRequirements, the *formatProperties* member is an instance of the VkSparseImageFormatProperties structure, the definition of which is:

```
typedef struct VkSparseImageFormatProperties {
    VkImageAspectFlags          aspectMask;
    VkExtent3D                  imageGranularity;
    VkSparseImageFormatFlags    flags;
} VkSparseImageFormatProperties;
```

The *aspectMask* member of VkSparseImageFormatProperties specifies the image aspect or aspects to which the remainder of the properties apply. This is a bitfield made up from members of the VkImageAspectFlagBits enumeration, the definition of which is:

```
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;
```

For each unique aspect of the image, an instance of `VkSparseImageFormatProperties` is returned. The *imageGranularity* member specifies the size, in texels, of the smallest region that may be uniquely bound within the image specified by *image*. Binding is affected by calling `vkQueueBindSparse`. It is an instance of the `VkExtent3D` structure which contains the size of the sparse binding regions, expressed in texels.

The *flags* member is a bitfield made up from members of the `VkSparseImageFormatFlagBits` enumeration and describes additional requirements for sparse memory binding. The definition of which is:

```
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;
```

The meanings of each of the flags is as follows:

If `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` is set, the image combines mulitple levels at the tail of the mip chain into a single residency state for array textures. Otherwise, mip tail is individually addressable for each array layer.

If `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` is set, then each mip level outside the tail has dimensions that are integer multiples of the sparse image block dimensions.

If `VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT` is set, then the format has non-standard sparse image block dimensions and the members of the *imageGranularity* do not match the standard sparse image block dimensions for the format.

The *imageMipTailStartLod* member of the `VkSparseImageMemoryRequirements` structure contains the level-of-detail at which the mip tail begins for the image specified in *image*.

imageMipTailSize contains the size of the mip tail, *imageMipTailOffset* contains its offset, and *imageMipTailStride* contains the stride between layers in the

tail for array textures. All three members are expressed in bytes.

Valid Usage

- *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*
- Each of *device* and *image* must have been created, allocated or retrieved from the same `VkPhysicalDevice`

1.107.5 See Also

`vkQueueBindSparse`, `vkGetImageMemoryRequirements`

1.108 vkGetImageSubresourceLayout(3)

1.108.1 Name

vkGetImageSubresourceLayout - Retrieve information about an image subresource.

1.108.2 C Specification

```
void vkGetImageSubresourceLayout(
    VkDevice                                    device,
    VkImage                                     image,
    const VkImageSubresource*                   pSubresource,
    VkSubresourceLayout*                        pLayout);
```

1.108.3 Parameters

device
A handle to the device that owns the image.

image
A handle to the image about which to retrieve information.

pSubresource
A pointer to a structure describing the image subresource.

pLayout
A pointer to a structure that will receive information about the image subresource.

1.108.4 Description

vkGetImageSubresourceLayout returns information about the memory layout of an image subresource of an image. *device* is a handle to the device that owns *image*, which is the image about which to retrieve information. A description of the image subresource is passsed to the command through an instance of the VkImageSubresource structure, the address of which is passed in *pSubresource*. The definition of VkImageSubresource is:

```
typedef struct VkImageSubresource {
    VkImageAspectFlags    aspectMask;
    uint32_t              mipLevel;
    uint32_t              arrayLayer;
} VkImageSubresource;
```

Within the *pSubresource* structure, *aspectMask* is a bitfield describing the aspect of the image and is made up of a single member of the VkImageAspectFlagBits enumeration, the definition of which is:

```
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 VK_IMAGE_ASPECT_COLOR_BIT aspect is valid only for image formats that are usable as color. The VK_IMAGE_AS PECT_DEPTH_BIT aspect is valid for formats containing depth information and the VK_IMAGE_ASPECT_STENCIL_BIT aspect is valid only for formats containing stencil information. Note that some formats contain both depth and stencil information, and in this case, *aspectMask* is used to select which to query. It is not legal to include more than one member of VkImageAsp ectFlagBits in *aspectMask*. Some formats also include metadata which may be implementation dependent but is queryable by specifying VK_IMAGE_ASPECT_METADATA_BIT.

For resources that have mipmaps or are multiple array layers, the *mipLevel* and *arrayLayer* members describe the mipmap level and array layer, respectively. For resources that do not have mipmaps or are not layered, *mipLevel* and *arrayLayer*, respectively, should be set to zero.

Information about the selected sub-resource is returned to the caller in the instance of the VkSubresourceLayout structure pointed to by *pLayout*. The definition of VkSubresourceLayout is:

```
typedef struct VkSubresourceLayout {
    VkDeviceSize    offset;
    VkDeviceSize    size;
    VkDeviceSize    rowPitch;
    VkDeviceSize    arrayPitch;
    VkDeviceSize    depthPitch;
} VkSubresourceLayout;
```

The *offset* member of the *pLayout* structure is filled with the relative offset of the start of the sub-resource from the start of the parent resource's memory binding. The *size* member is the size of the sub-resource in memory. The *rowPitch* specifies the distance in memory of the start of each texel row of the sub-resource from the start of the previous row. The *depthPitch* specifies the distance in memory of the start of each slice of the sub-resource relative to the start of the previous slice. For one-dimensional resources, *rowPitch* is zero, and for one- and two-dimensional resources, *depthPitch* is zero. All four parameters are specified in bytes.

Valid Usage

- *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*
- Each of *device* and *image* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *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

1.108.5 See Also

vkGetPhysicalDeviceImageFormatProperties

1.109 vkGetInstanceProcAddr(3)

1.109.1 Name

vkGetInstanceProcAddr - Return a function pointer for a command

1.109.2 C Specification

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

1.109.3 Parameters

instance
The instance whose function pointer to query

pName
The name of the command

1.109.4 Description

vkGetInstanceProcAddr returns a function pointer for the command specified in *pName* as it corresponds to *instance*. Depending on the operating system, supporting components, software environment and hardware topology, the address returned for a single command name may be different for different values of *instance*.

If *instance* is **NULL**, **vkGetInstanceProcAddr** will return non-**NULL** function pointers for the global commands **vkEnumerateInstanceExtensionProperties**, **vkEnumerateInstanceLayerProperties**, and **vkCreateInstance**. It will return **NULL** for all other commands, since they may have different implementations in different instances.

If *instance* is a valid instance, **vkGetInstanceProcAddr** will return a non-**NULL** function pointer for any core command except the global commands listed previously. It will also return non-**NULL** for any extension command, if there is a layer or driver available that implements the extension.

The function pointers returned by **vkGetInstanceProcAddr** may be used with any object of the appropriate type derived from the *instance*. For example, the function pointer for a command with a VkDevice first parameter can be used with any VkDevice object created from physical devices belonging to the instance.

Valid Usage

- If *instance* is not NULL, *instance* must be a valid VkInstance handle
- *pName* must be a null-terminated string
- If *instance* is NULL, *pName* must be one of: **vkEnumerateInstanceExtensionProperties**, **vkEnumerateInstanceLayerProperties** or **vkCreateInstance**
- If *instance* is not NULL, *pName* must be the name of a core command or a command from an enabled extension, other than: **vkEnumerateInstanceExtensionProperties**, **vkEnumerateInstanceLayerProperties** or **vkCreateInstance**

1.109.5 Return Value

Upon success, **vkGetInstanceProcAddr** returns the address (**PFN_vkVoidFunction**) of the command whose name is specified by *pName*. If *pName* is not supported by the Vulkan library, then **vkGetInstanceProcAddr** returns **NULL**.

1.109.6 See Also

vkGetDeviceProcAddr, vkCreateInstance

1.110 vkGetPhysicalDeviceFeatures(3)

1.110.1 Name

vkGetPhysicalDeviceFeatures - Reports capabilities of a physical device.

1.110.2 C Specification

```
void vkGetPhysicalDeviceFeatures(
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceFeatures*                   pFeatures);
```

1.110.3 Parameters

physicalDevice
A handle to the physical device.

pFeatures
A pointer to a structure that will be written with the device feature set.

1.110.4 Description

vkGetPhysicalDeviceFeatures returns the set of physical features supported by the physical device whose handle is passed in *physicalDevice*. This parameter should be a valid handle to a physical device returned from a successful call to vkEnumeratePhysicalDevices. *pFeatures* is a pointer to an instance of the VkPhysicalDeviceFeatures structure, the definition of which is:

```
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;
```

Each member of the *pFeatures* structure represents a feature of the underlying physical device. A brief description of the members follows:

- ***robustBufferAccess*** indicates that out of bounds accesses to buffers via shader operations are well-defined.
  - When enabled, out-of-bounds buffer reads will return any of the following values:
    - Values from anywhere within the buffer object.
    - 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
  - When enabled, out-of-bounds writes may modify values within the buffer object or be ignored.
  - If not enabled, out of bounds accesses may cause undefined behaviour up-to and including process 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. The *maxDrawIndexedIndexValue* limit indicates 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 [?].
- ***imageCubeArray*** indicates whether image views with a VkImageViewType of VK_IMAGE_VIEW_TYPE_CUBE_ARRAY can be created and that the corresponding **ImageCubeArray** SPIR-V OpCapability can be used in shader code.
- ***independentBlend*** indicates whether the VkPipelineColorBlendAttachmentState settings are controlled independently per-attachment. If this is features not supported or enabled, the VkPipelineColorBlendAttachmentState settings for the first color attachment will be used for all attachments. Otherwise, a VkPipelineColorBlendAttachmentState must be provided for each bound color attachment.
- ***geometryShader*** indicates whether geometry shaders are supported. If this feature is not supported or enabled, VK_SHADER_STAGE_GEOMETRY_BIT, and VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT enum values may not be used. This also indicates whether the **Geometry** SPIR-V OpCapability can be used in shader code.

- ***tessellationShader*** indicates whether tessellation control and evaluation shaders are supported. If this feature is not supported or 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 may not be used. This also indicates whether the **Tessellation** SPIR-V OpCapability can be used in shader code.

- ***sampleRateShading*** indicates whether per-sample shading and multisample interpolation is supported. If this feature is not supported or enabled, the *sampleShadingEnable* member of the VkPipelineMultisampleStateCreateInfo structure must be set to VK_FALSE and the *minSampleShading* member is ignored. This also indicates whether the **SampleRateShading** SPIR-V OpCapability can be used in shader code.

- ***dualSourceBlend*** indicates whether blend operations which take two sources are supported. If this feature is not supported or 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 may not be used as source or destination blending factors.

- ***logicOp*** indicates whether logic operations are supported. If this feature is not supported or enabled, the *logicOpEnable* member of the VkPipelineColorBlendStateCreateInfo structure must be set to VK_FALSE and the *logicOp* member is ignored.

```
[[features-features-multiDrawIndirect]]
* *pname:multiDrawIndirect* indicates whether multi-draw indirect is
  supported. If this feature is not supported or enabled, the
  ptext:drawCount parameter to the flink:vkCmdDrawIndirect and
  flink:vkCmdDrawIndexedIndirect commands must be 1. The
  ptext:maxDrawIndirectCount member of the slink:VkPhysicalDeviceLimits
  structure must also be 1 if this feature is not supported. See
  <<features-limits-maxDrawIndirectCount>>.
```

- ***depthClamp*** indicates whether depth clamping is supported. If this feature is not supported or 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 supported or enabled, the *depthBiasClamp* parameter to vkCmdSetDepthBias is ignored.

- ***fillModeNonSolid*** indicates whether point and wireframe fill modes are supported. If this feature is not supported or enabled, the VK_POLYGON_MODE_POINT and VK_POLYGON_MODE_LINE enum values may not be used.

- ***depthBounds*** indicates whether depth bounds tests are supported. If this feature is not supported or enabled, the *depthBoundsTestEnable* member of the VkPipelineDepthStencilStateCreateInfo structure must be set to VK_FALSE. When *depthBoundsTestEnable* is set to VK_FALSE, the values of the vkCmdSetDepthBounds command may not be used.

- ***wideLines*** indicates whether lines with width greater than 1.0 are supported. If this feature is not supported or enabled, the vkCmdSetLineWidth command may not be used.

  1. The range and granularity of supported line widths are indicated by the *lineWidthRange* and *lineWidthGranularity* members of the VkPhysicalDeviceLimits structure, respectively.

- ***largePoints*** indictates if points with size greater than 1.0 are supported. If this feature is not supported or 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.

- ***textureCompressionETC2*** indicates whether the ETC2 and EAC compressed texture formats are supported. If this feature is not supported or enabled, the following formats may not be used to create images:

  - 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 **vkGetPhysicalDeviceFormatProperties** command should be used to check for the supported properties of individual formats.

- ***textureCompressionASTC_LDR*** indicates whether the ASTC LDR compressed texture formats are supported. If this feature is not supported or enabled, the following formats may not be used to create images:

- 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 **vkGetPhysicalDeviceFormatProperties** command should be used to check for the supported properties of individual formats.

- ***textureCompressionBC*** indicates whether the BC compressed texture formats are supported. If this feature is not supported or enabled, the following formats may not be used to create images:
  - 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 **vkGetPhysicalDeviceFormatProperties** command should be used to check for the supported properties of individual formats.

- ***occlusionQueryPrecise*** indicates whether precise (non-conservative) occlusion queries 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 supported and enabled, queries of this type may set VK_QUERY_CONTROL_PRECISE_BIT in the *flags* parameter to **vkCmdBeginQuery**. If this feature is not supported, the implementation can only support conservative occlusion queries. When any samples are passed, conservative queries will return between one and the actual number of samples passed. 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 supported or enabled, queries of type VK_QUERY_TYPE_PIPELINE_STATISTICS cannot be created and none of the VkQueryPipelineStatisticFlagBits bits should 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 supported or enabled, all storage image, storage texel buffers and storage buffer variables in shaders for these stages must be decorated with the **NonWriteable** SPIR-V 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 supported or enabled, all storage image, storage texel buffers and storage buffer variables in shaders for the fragment stage must be decorated with the **NonWriteable** SPIR-V decoration (or the *readonly* memory qualifier in GLSL).

- *shaderTessellationAndGeometryPointSize* indicates whether the *PointSize* shader builtin is available in the tessellation control, tessellation evaluation, and geometry shader stages. If this feature is not supported or enabled, the *PointSize* shader builtin is not available in these shader stages and all points written from a tessellation or geometry shader will have a size of 1.0. This also indicates whether the **TessellationPointSize** SPIR-V OpCapability can be used in shader code for tessellation control and evaluation shaders, or if the **GeometryPointSize** SPIR-V OpCapability can be used in shader code 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 supported or enabled, the *textureGatherOffset* shader instruction only supports offsets that are constant integer expressions and the *textureGatherOffsets* shader instruction is not supported. This also indicates whether the **ImageGatherExtended** SPIR-V OpCapability can be used in shader code.

- ***shaderStorageImageExtendedFormats*** indicates whether the extended storage image formats are available in shader code. If this feature is not supported or enabled, the formats requiring the **StorageImageExtendedFormats** SPIR-V OpCapability are not supported for resources referenced by the `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` descriptor type. This also indicates whether the **StorageImageExtendedFormats** OpCapability ca be used in shader code.

- ***shaderStorageImageMultisample*** indicates whether multisampled storage images are supported. If this feature is not supported or enabled, images that are created with a *usage* that includes `VK_IMAGE_USAGE_STORAGE_BIT` must be created with *samples* equal to 1. This also indicates whether the **StorageImageMultisample** SPIR-V OpCapability can be used in shader code.

- ***shaderUniformBufferArrayDynamicIndexing*** indicates whether arrays of uniform buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not supported or enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` can only be indexed by constant integral expressions when aggregated into arrays in shader code. This corresponds to the **UniformBufferArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.

- ***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 supported or enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` and `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` can only be indexed by constant integral expressions when aggregated into arrays in shader code. This also indicates whether the **SampledImageArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.

- ***shaderStorageBufferArrayDynamicIndexing*** indicates whether arrays of storage buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not supported or enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` can only be indexed by constant integral expressions when aggregated into arrays in shader code. This corresponds to the **StorageBufferArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.

- ***shaderStorageImageArrayDynamicIndexing*** indicates whether arrays of storage images can be indexed by dynamically uniform integer expressions in shader code. If this feature is not supported or enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` can only be indexed by constant integral expressions when aggregated into arrays in shader code. This also indicates whether the **StorageImageArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.

- ***shaderClipDistance*** indicates whether clip distances are supported in shader code. If this feature is not supported or enabled, the *ClipDistance* shader builtin is not available in the builtin shader input and output blocks. This also indicates whether the **ClipDistance** SPIR-V OpCapability can be used in shader code.

- ***shaderCullDistance*** indicates whether cull distances are suppored in shader code. If this feature is not supported or enabled, the *CullDistance* shader builtin is not available in the builtin shader input and output blocks. This also indicates whether the **CullDistance** SPIR-V OpCapability can be used in shader code.

- ***shaderFloat64*** indicates whether 64-bit floats (doubles) are supported in shader code. If this feature is not supported or enabled, the 64-bit floating point types cannot be used in shader code. This also indicates whether the **Float64** SPIR-V OpCapability can be used in shader code.

- ***shaderInt64*** indicates whether 64-bit integers (signed and unsigned) are supported in shader code. If this feature is not supported or enabled, the 64-bit integer types cannot be used in shader code. This also indicates whether the **Int64** SPIR-V OpCapability can be used in shader code.

- ***shaderInt16*** indicates whether 16-bit integers (signed and unsigned) are supported in shader code. If this feature is not supported or enabled, the 16-bit integer types cannot be used in shader code. This also indicates whether the **Int16** SPIR-V OpCapability can be used in shader code.

- ***shaderResourceResidency*** indicates whether image operations that return resource residency information are supported in shader code. If this feature is not supported or enabled, the image operations which return resource residency information cannot be used in shader code. This also indicates whether the **SparseResidency** SPIR-V OpCapability can be used in shader code. The feature requires the *sparseNonResident* feature 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 supported or enabled, the image operations which specify minimum resource LOD cannot be used in shader code. This also indicates whether the **MinLod** SPIR-V OpCapability can be used in shader code.

- ***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 supported or enabled, then the *alphaToOneEnable* member of the VkPipelineColorBlendAttachmentState structure must be set to VK_FALSE. Otherwise setting *alphaToOneEnable* to VK_TRUE will enable alpha-to-one behaviour.

- ***sparseBinding*** indicates whether resource memory can be managed at opaque page level instead of at the object level. If this feature is not supported or enabled, resource memory can only be bound on a per-object basis using the **vkBindBufferMemory** and **vkBindImageMemory** commands. In this case, buffers and images cannot 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 supported or enabled, buffers cannot 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 supported or enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* of 1 cannot 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 supported or enabled, images with an *imageType* of VK_IMAGE_TYPE_3D cannot 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 supported or enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* of 2 cannot 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 supported or enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* of 4 cannot 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 supported or enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* of 8 cannot 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 supported or enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* of 16 cannot 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 supported or enabled, the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT enum values may not be used in *flags* members of the VkBufferCreateInfo and VkImageCreateInfo structures, respectively.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *pFeatures* must be a pointer to a VkPhysicalDeviceFeatures structure

1.110.5 See Also

vkGetPhysicalDeviceProperties

1.111 vkGetPhysicalDeviceFormatProperties(3)

1.111.1 Name

vkGetPhysicalDeviceFormatProperties - Lists physical device's format capabilities.

1.111.2 C Specification

```
void vkGetPhysicalDeviceFormatProperties(
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkFormatProperties*                         pFormatProperties);
```

1.111.3 Parameters

physicalDevice
A handle to the physical device to query.

format
The format whose properties to query.

pFormatProperties
A pointer to the structure to receive the result of the query.

1.111.4 Description

vkGetPhysicalDeviceFormatProperties queries the device specified by *physicalDevice* for its support of the format specified in *format* and places the result in the structure pointed to by *pFormatProperties*. *pFormatProperties* should point to an instance of the VkFormatProperties structure, the definition of which is:

```
typedef struct VkFormatProperties {
    VkFormatFeatureFlags    linearTilingFeatures;
    VkFormatFeatureFlags    optimalTilingFeatures;
    VkFormatFeatureFlags    bufferFeatures;
} VkFormatProperties;
```

The *linearTilingFeatures*, *optimalTilingFeatures*, and *bufferFeatures* parameters are a bitwise combination of one or more of the bits specified in VkFormatFeatureFlagBits, the definition of which is:

```
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,
} VkFormatFeatureFlagBits;
```

The *linearTilingFeatures* member contains information about format support in linear images. The *optimalTilingFeatures* member contains information about format support in opaque tiled images. The *bufferFeatures* member contains information about format support in buffer objects.

- If VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT is set then image views of this format may be sampled by shaders. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.
- If VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT is set then image views of this format may be used as storage images in shaders. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.
- If VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT is set is set then atomic operations may be performed by shaders on image views of this format. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.
- If VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT is set then buffer views of this format may be used to store uniform values accessible by shaders. This bit should only appear in *bufferFeatures*.
- If VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT is set then buffer views of this format may be used to store texel data accessible to shaders. This bit should only appear in *bufferFeatures*.
- If VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT is set then shaders may perform atomic operations on texel data stored in buffers in this format. This bit should only appear in *bufferFeatures*.
- If VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT is set then this format may be used as a vertex attribute format. This bit should only appear in *bufferFeatures*.
- If VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT is set then image views of this format may be used as color attachments. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.
- If VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT is set then blending is supported into color attachments of this format. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.
- If VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT is set then image views of this format may be used as depth or stencil attachments. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.
- If VK_FORMAT_FEATURE_BLIT_SRC_BIT is set then images of this format may be used as the source of a blit operation. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.
- If VK_FORMAT_FEATURE_BLIT_DST_BIT is set then images of this format may be used as the destination of a blit operation. This bit should only appear in *linearTilingFeatures* or *optimalTilingFeatures*.

If the physical device does not support the specified format then the output VkFormatProperties structure is filled with zeros.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *format* must be a valid VkFormat value
- *pFormatProperties* must be a pointer to a VkFormatProperties structure

1.111.5 See Also

vkGetPhysicalDeviceImageFormatProperties, vkGetPhysicalDeviceFeatures

1.112 vkGetPhysicalDeviceImageFormatProperties(3)

1.112.1 Name

vkGetPhysicalDeviceImageFormatProperties - Lists physical device's image format capabilities.

1.112.2 C Specification

```
VkResult vkGetPhysicalDeviceImageFormatProperties(
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkImageType                                 type,
    VkImageTiling                               tiling,
    VkImageUsageFlags                           usage,
    VkImageCreateFlags                          flags,
    VkImageFormatProperties*                    pImageFormatProperties);
```

1.112.3 Parameters

physicalDevice
: A handle to the physical device upon which to perform the query.

format
: The format of the image.

type
: The type of the image.

tiling
: The tiling mode of the image.

usage
: The usage of the image.

flags
: Additional flags describing the image.

pImageFormatProperties
: A pointer to a structure in which the requested information is returned.

1.112.4 Description

vkGetPhysicalDeviceImageFormatProperties queries the physical device specified in *physicalDevice* about its support for images as if they had been created using the remaining parameters to the command. These parameters, *format*, *type*, *tiling* and *usage* have the same meanings as they do in the VkImageCreateInfo structure, the definition of which is:

```
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;
```

In this call, *format* specifies the format of the image and must be a member of the VkFormat enumeration. *type* specifies the type of image for which the format will be used. This is a member of the VkImageType enumeration, the definition of which is:

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

The *tiling* parameter specifies the tiling layout for the image and must be one of VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL. The *usage* parameter specifies the intended usage for the image and should be a bitwise combination of one or more members of the VkImageUsageFlagBits enumeration, the definition of which is:

```
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;
```

If the image format is supported for the specified combination of the *format*, *type*, *tiling*, *usage*, and *flags* parameters, then **vkGetPhysicalDeviceImageFormatProperties** places information about how such an image may be used in the instance of the VkImageFormatProperties structure pointed to by *pImageFormatProperties*. The definition of VkImageFormatProperties is:

```
typedef struct VkImageFormatProperties {
    VkExtent3D            maxExtent;
    uint32_t              maxMipLevels;
    uint32_t              maxArrayLayers;
    VkSampleCountFlags    sampleCounts;
    VkDeviceSize          maxResourceSize;
} VkImageFormatProperties;
```

The *maxExtent* member of the output structure contains the maximum dimensions of an image in the specified format. The *maxMipLevels* and *maxArrayLayers* contain the maximum number of mipmap levels and maximum number of layers in array forms of images, respectively. If array images are not supported for the specified format, then *maxArrayLayers* will be zero. If multisampling is supported for the specified format, then *sampleCounts* contains a bitwise combination of the supported sample counts using members of the VkSampleCountFlagBits enumeration, the definition of which is:

```
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;
```

Additional sample counts not listed in the `VkSampleCountFlagBits` enumeration are reserved and their presence should not be used to imply higher sample counts than listed.

If multisampling is not supported for the specified format, then *sampleCounts* will contain zero.

The *maxResourceSize* member contains the maximum size in memory of any resource in this format. Note that it may not be possible to create a resource of the maximum supported extent and array layer count in every dimension that still fits within the maxium in-memory resource size.

If the format is not supported in the specified configuration, then **vkGetPhysicalDeviceImageFormatProperties** fills the `VkImageFormatProperties` structure pointed to by *pImageFormatProperties* with zeros.

Valid Usage

- *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`

1.112.5 See Also

`vkGetPhysicalDeviceProperties`, `vkGetPhysicalDeviceFeatures`

1.113 vkGetPhysicalDeviceMemoryProperties(3)

1.113.1 Name

vkGetPhysicalDeviceMemoryProperties - Reports memory information for the specified physical device.

1.113.2 C Specification

```
void vkGetPhysicalDeviceMemoryProperties(
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceMemoryProperties*           pMemoryProperties);
```

1.113.3 Parameters

physicalDevice
A handle to the physical device about which information is to be retrieved.

pMemoryProperties
A pointer to a structure that is to receive the memory information.

1.113.4 Description

vkGetPhysicalDeviceMemoryProperties retrieves information about the memory of the physical device whose handle is given in *physicalDevice*. *pMemoryProperties* should point to an instance of the VkPhysicalDeviceMemoryProperties structure, into which will be stored the information about the device. The definition of VkPhysicalDeviceMemoryProperties is as follows.

```
typedef struct VkPhysicalDeviceMemoryProperties {
    uint32_t        memoryTypeCount;
    VkMemoryType    memoryTypes[VK_MAX_MEMORY_TYPES];
    uint32_t        memoryHeapCount;
    VkMemoryHeap    memoryHeaps[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryProperties;
```

- *memoryTypeCount* will be filled with the number of memory types supported by the device.
- *memoryTypes* is an array of VK_MAX_MEMORY_TYPES instances of the VkMemoryType structures. Upon return from **vkGetPhysicalDeviceMemoryProperties**, the first *memoryTypeCount* elements of *memoryTypes* will contain valid data.
- *memoryHeapCount* will be filled with the number of memory heaps supported by the device.
- *memoryHeaps* is an array of VK_MAX_MEMORY_HEAPS VkMemoryHeap structures which describe the heaps available to the device.

The definition of the VkMemoryType structure is as follows:

```
typedef struct VkMemoryType {
    VkMemoryPropertyFlags    propertyFlags;
    uint32_t                 heapIndex;
} VkMemoryType;
```

- *propertyFlags* is a bitfield made up from members of the VkMemoryPropertyFlagBits enumeration, which is described below.

- *heapIndex* is the index into the *memoryHeaps* array returned through this command from which the memory type will be allocated.

The VkMemoryPropertyFlagBits enumeration, which forms the available bits for use in the *propertyFlags* member of the VkMemoryType structure is defined as follows:

```
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;
```

- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT signifies that the memory is the most efficient type for device access (e.g. local device memory).
- VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT indicates that memory with this property is visible to the host. That is, a valid host address may be obtained and allocations from this memory type may be mapped.
- VK_MEMORY_PROPERTY_HOST_COHERENT_BIT indicates that accesses to mapped memory of this type is coherent with accesses to the same memory by the device. Such access do not need to be marshalled using calls to vkFlushMappedMemoryRanges or by unmapping the memory.
- VK_MEMORY_PROPERTY_HOST_CACHED_BIT indicates that data stored in memory of this type is cached by the host and as such, it is likely that reads from such regions by the host will be faster than reads from uncached memory.
- VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT indicates that allocations from this type of memory may be made on an as-needed basis. In general, allocations from this type of memory will almost always succeed and return quickly, but first access to such a region may take longer than expected.

The definition of the VkMemoryHeap structure is as follows:

```
typedef struct VkMemoryHeap {
    VkDeviceSize         size;
    VkMemoryHeapFlags    flags;
} VkMemoryHeap;
```

- *size* specifies the size in bytes of the memory heap.
- *flags* is a bitfield made up of the members of the VkMemoryHeapFlagBits enumeration, the definition of which is:

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

- If the *flags* member of VkMemoryHeap contains VK_MEMORY_HEAP_HOST_LOCAL_BIT, then the memory for that heap is located closer to the host than to the device in NUMA (Non-Unified Memory Architecture) systems. Even in unified architectures, this flag may indicate that access to this heap is more efficient from the host than from the device.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *pMemoryProperties* must be a pointer to a VkPhysicalDeviceMemoryProperties structure

1.113.5 See Also

vkGetPhysicalDeviceProperties

1.114 vkGetPhysicalDeviceProperties(3)

1.114.1 Name

vkGetPhysicalDeviceProperties - Returns properties of a physical device.

1.114.2 C Specification

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

1.114.3 Parameters

physicalDevice
A handle to the physical device.

pProperties
A pointer to a structure that will be written with the device properties.

1.114.4 Description

vkGetPhysicalDeviceProperties returns the properties of the physical device specified in *physicalDevice* in the structure pointed to by *pProperties*. *pProperties* points to an instance of the VkPhysicalDeviceProperties structure, the definition of which is:

```
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;
```

The data returned in the *pProperties* structure contains information about the physical device and the driver associated with it.

The *apiVersion* member of VkPhysicalDeviceProperties indicates the API version supported by the physical device. Minor revisions of the API are backward compatible whereas major versions of the API may break compatibility. The API version is represented as a 32-bit field where bits 31 - 22 represent the major version, bits 21 - 12 represent the minor version, and bits 11 - 0 represent the patch version.

The *driverVersion* member represents the vendor-specific version of the driver used to enable the device.

The *vendorID* and *deviceID* members contain the PCI vendor and device identifiers, respectively. Note that if the device is not physcially a PCI-compliant device, then the values of *vendorID* and *deviceID* are platform dependent and may not be values assigned by the PCI-SIG.

The *deviceType* member indicates the type of device represented by *physicalDevice*. *deviceType* is a member of the VkPhysicalDeviceType enumeration, the definition of which is:

```
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;
```

When *deviceType* is VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU, then the device is typically one embedded in or tightly coupled with the host CPU that is running the application. When *deviceType* is VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU, then the device is typically a separate physical device connected to the host CPU via a slower interlink such as PCI-Express. If *deviceType* is VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU, then the device is potentially emulated (such a stub device or debugger endpoint), a virtual node in a virtualization environment or otherwise does not fit either category. If the device is running entirely on the CPU, then *deviceType* will be VK_PHYSICAL_DEVICE_TYPE_CPU. If the device type is unknown or does not fit any of these types, then *deviceType* may be VK_PHYSICAL_DEVICE_TYPE_OTHER.

The *deviceName* member of *pProperties* contains a vendor-supplied human-readable name for the device encoded as a UTF-8 string which is up to VK_MAX_PHYSICAL_DEVICE_NAME_SIZE characters long, including a null-terminator.

pipelineCacheUUID is an array, of size VK_UUID_SIZE, containing 8-bit values that represent a universally unique signature that identifies the hardware and driver combination.

limits is an instance of the VkPhysicalDeviceLimits structure which contains limits on the functionality provided by the device. The definition of VkPhysicalDeviceLimits is:

```
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;
```

Valid Usage

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

1.114.5 See Also

`vkGetPhysicalDeviceFeatures`, `vkGetPhysicalDeviceProperties`

1.115 vkGetPhysicalDeviceQueueFamilyProperties(3)

1.115.1 Name

vkGetPhysicalDeviceQueueFamilyProperties - Reports properties of the queues of the specified physical device.

1.115.2 C Specification

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

1.115.3 Parameters

physicalDevice
Physical device to query.

pQueueFamilyPropertyCount
Count indicating number of `VkQueueFamilyProperties` pointed to by *pQueueFamilyProperties*.

pQueueFamilyProperties
Pointer to an array of `VkQueueFamilyProperties` structures receiving the information about each particular queue family.

1.115.4 Description

vkGetPhysicalDeviceQueueFamilyProperties retrieves properties of the queues on a physical device whose handle is given in *physicalDevice*. *pQueueFamilyPropertyCount* must be set to the size of the array pointed to by *pQueueFamilyProperties* and thus specifies the number of queue families to retrieve information for. The *pQueueFamilyProperties* parameter should point to an array of `VkQueueFamilyProperties` structures to be filled out with the properties of the queue families. If *pQueueFamilyProperties* is **NULL** then **vkGetPhysicalDeviceQueueFamilyProperties** will update the value pointed by *pQueueFamilyPropertyCount* with the number of queue families available on the specified physical device.

The device will overwrite the entries of *pQueueFamilyProperties* with information about the supported queues, and will write the number of structures filled into the variable pointed to by *pQueueFamilyPropertyCount*. Each element of *pQueueFamilyProperties* is an instance of the `VkQueueFamilyProperties` structure, the definition of which is:

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

The members of `VkQueueFamilyProperties` have the following meanings:

- *queueFlags* is a bitfield made up from members of the `VkQueueFlagBits` enumeration indicating capabilities of the queue. The list of capabilities is described below.
- *queueCount* contains the number of individual queues within the specified queue family. Queues within a single family are considered identical from a feature support perspective and are directly compatible with one another.

- *timestampValidBits* contains the number of valid bits that will be written to timestamp by `vkCmdWriteTimestamp`. Timestamps are always 64-bit unsigned integers. However, less than 64 bits may actually be valid. Additional bits will contain zeros. If *timestampValidBits* is zero then the queue does not support timestamps and `vkCmdWriteTimestamp` may not be used in command buffers submitted to queues in this family. If *timestampValidBits* is non-zero, it must be at least 32, and may be as high as 64.

The valid bits that may be contained in *queueFlags* are comprised of the members of the `VkQueueFlagBits` enumeration, the definition of which is:

```
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 a queue's *queueFlags* member contains `VK_QUEUE_GRAPHICS_BIT`, then it supports graphics operations such as binding graphics state and graphics pipelines and executing drawing commands.
- If a queue's *queueFlags* member contains `VK_QUEUE_COMPUTE_BIT`, then it supports compute operations such as binding compute pipelines and executing compute dispatches.
- If a queue's *queueFlags* member contains `VK_QUEUE_TRANSFER_BIT`, then it supports transfer operations, which include copying data and images.
- If a queue's *queueFlags* member contains `VK_QUEUE_SPARSE_BINDING_BIT`, then it supports binding memory to sparse buffer and image resources.

All Vulkan implementations must expose at least one queue, each of which has at least one queue capability bit set. Note, though, that it is possible that new sets of capabilities are exposed by extensions or future API versions and so a queue may have none of the bits listed above set.

Valid Usage

- *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

1.115.5 See Also

`vkGetPhysicalDeviceFeatures`, `vkGetPhysicalDeviceProperties`, `vkGetPhysicalDeviceMemoryProperties`, `VkQueueFamilyProperties`

1.116 vkGetPhysicalDeviceSparseImageFormatProperties(3)

1.116.1 Name

vkGetPhysicalDeviceSparseImageFormatProperties - Retrieve properties of an image format applied to sparse images.

1.116.2 C Specification

```
void vkGetPhysicalDeviceSparseImageFormatProperties(
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkImageType                                 type,
    VkSampleCountFlagBits                       samples,
    VkImageUsageFlags                           usage,
    VkImageTiling                               tiling,
    uint32_t*                                   pPropertyCount,
    VkSparseImageFormatProperties*              pProperties);
```

1.116.3 Parameters

physicalDevice
The physical device whose properties to query.

format
The format about which to query the device.

type
The dimensionality of the image.

samples
The number of multisamples in the image.

usage
The intended usages for the image.

tiling
A set of flags defining the tiling of the image.

pPropertyCount
A pointer to a variable that contains the number of properties to query.

pProperties
A pointer to an array of `VkSparseImageFormatProperties` structures that will receive the results of the query.

1.116.4 Description

`vkGetPhysicalDeviceSparseImageFormatProperties` queries the physical device specified in *`physicalDevice`* for its support of the format described by the remaining parameters to the command should that format be used with a sparse image.

`format` specifies the format of the image and is a member of the `VkFormat` enumeration. *`type`* specifies the type of the image and is a member of the `VkImageType` enumeration, the definition of which is:

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

The `samples` parameter specifies the number of samples to be used in the image and must be a supported sample count for the image format. The possible values are:

```
typedef enum VkSampleCountFlagBits {
    VK_SAMPLE_COUNT_1_BIT = 0x00000001,
    VK_SAMPLE_COUNT_2_BIT = 0x00000002,
    VK_SAMPLE_COUNT_4_BIT = 0x00000004,
    VK_SAMPLE_COUNT_8_BIT = 0x00000008,
    VK_SAMPLE_COUNT_16_BIT = 0x00000010,
    VK_SAMPLE_COUNT_32_BIT = 0x00000020,
    VK_SAMPLE_COUNT_64_BIT = 0x00000040,
} VkSampleCountFlagBits;
```

If `samples` is `VK_SAMPLE_COUNT_1_BIT` then the image is not multisampled.

The `usage` parameter is a bitfield made up of members of the `VkImageUsageFlagBits` enumeration and specifies the intended usage for the image. The definition of `VkImageUsageFlagBits` is:

```
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;
```

On input, `pPropertyCount` points to a variable that is populated with the number of aspects to query about the image. `pProperties` is a pointer to an array of at least this many `VkSparseImageFormatProperties` structures, the definition of which is:

```
typedef struct VkSparseImageFormatProperties {
    VkImageAspectFlags          aspectMask;
    VkExtent3D                  imageGranularity;
    VkSparseImageFormatFlags    flags;
} VkSparseImageFormatProperties;
```

On return, the variable pointed to by `pPropertyCount` will be overwritten with the number of entries in `pProperties` that were populated by the command.

In the `VkSparseImageFormatProperties` structure, `aspectMask` contains the aspects of the image to which this property applies. `imageGranularity` contains the size, in texels at which image memory is to be bound to a sparse image with the specified properties through a call to `vkQueueBindSparse`. `flags` contains a bitfield of the supported flags for this image format, and is a bitwise combination of members of the `VkSparseImageFormatFlagBits` enumeration, the definition of which is:

```
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;
```

Valid Usage

- `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `format` must be a valid `VkFormat` value
- `pFormatProperties` must be a pointer to a `VkFormatProperties` structure

1.116.5 Return Value

vkGetPhysicalDeviceSparseImageFormatProperties does not return a value. However, on success, the variable *pPropertyCount* is overwritten with the number of structures written into the array pointed to by *pProperties*. On failure, the variable is overwritten with zero.

1.116.6 See Also

vkGetImageSparseMemoryRequirements, vkGetPhysicalDeviceImageFormatProperties, vkGetPhysicalDeviceFormatProperties

1.117 vkGetPipelineCacheData(3)

1.117.1 Name

vkGetPipelineCacheData - Get the data store from a pipeline cache

1.117.2 C Specification

```
VkResult vkGetPipelineCacheData(
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    size_t*                                     pDataSize,
    void*                                       pData);
```

1.117.3 Parameters

device
A handle to the device that is the parent of the pipeline cache.

pipelineCache
The pipeline cache whose data will be returned.

pDataSize
A pointer to a variable to receive the size (in bytes) of the data retrieved from the cache.

pData
A pointer to memory where the cache's data will be stored.

1.117.4 Description

vkGetPipelineCacheData fills the output buffer *pData* with a copy of the data store of a pipeline cache, as a step in the process of the application retrieving and saving the cache data. *pDataSize* points to a variable that, on entry, contains the size of the data area pointed to by *pData*. If *pData* is **NULL**, then the initial value of the variable addressed by *pDataSize* is ignored and overwritten with the size of the data that would be returned. Otherwise, it is used to determine the size of data that may be written to *pData*, which should be large enough to receive the entire data blob.

Valid Usage

- *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*
- Each of *device* and *pipelineCache* must have been created, allocated or retrieved from the same VkPhysicalDevice

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

1.117.5 See Also

vkCreatePipelineCache, vkMergePipelineCaches

1.118 vkGetQueryPoolResults(3)

1.118.1 Name

vkGetQueryPoolResults - Copy results of queries in a query pool to a host memory region.

1.118.2 C Specification

```
VkResult vkGetQueryPoolResults(
    VkDevice                                    device,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount,
    size_t                                      dataSize,
    void*                                       pData,
    VkDeviceSize                                stride,
    VkQueryResultFlags                          flags);
```

1.118.3 Parameters

device
Logical device owning the query pool.

queryPool
The query pool whose results should be copied to the buffer object.

startQuery
The index of the first query in the query pool whose results should be copied to the buffer object.

queryCount
The number of queries in the query pool whose results should be copied to the buffer object.

dataSize
The size of the data area pointed to by *pData*.

pData
A pointer to a buffer that will be filled with query results.

stride
The stride, in bytes between the start of each query object in memory.

flags
The flags controlling the behavior of the query result copy command (see `VkQueryResultFlags`).

1.118.4 Description

vkGetQueryPoolResults copies the results of *queryCount* number of queries in the query pool specified by *queryPool* starting from index *startQuery*. The results are written to the host memory buffer specified by *pData*. *dataSize* contains the size of the output buffer. If *pData* is not **NULL**, then the output buffer size must be large enough to hold the query results. The semantics of when and what values written to the destination buffer are defined by the type of the queries in the query pool, the query control flags passed to `vkCmdBeginQuery`, and the query result control flags specified by *flags*.

Valid Usage

- *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*
- Each of *device* and *queryPool* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *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

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

1.118.5 Return Value

Upon success, **vkGetQueryPoolResults** returns VK_SUCCESS and query results are deposited in the buffer pointed to by *pData*. If only some of the results are available, **vkGetQueryPoolResults** returns VK_NOT_READY; in this case, if *flags* contains VK_QUERY_RESULT_PARTIAL_BIT, partial results are deposited in the buffer pointed to by *pData*. Upon failure, a descriptive error code is returned.

1.118.6 See Also

vkCmdCopyQueryPoolResults, VkQueryResultFlags, vkCmdBeginQuery, VkQueryControlFlags

1.119 vkGetRenderAreaGranularity(3)

1.119.1 Name

vkGetRenderAreaGranularity - Returns the granularity for optimal render area.

1.119.2 C Specification

```
void vkGetRenderAreaGranularity(
    VkDevice                                    device,
    VkRenderPass                                renderPass,
    VkExtent2D*                                 pGranularity);
```

1.119.3 Parameters

device
: The device in which *renderPass* was created.

renderPass
: The render pass for which to query the render area granularity.

pGranularity
: A pointer to a structure containing the return value.

1.119.4 Description

vkGetRenderAreaGranularity returns the granularity at which the *renderArea* member of the VkRenderPassBeginInfo structure should be for optimal performance. *device* must be the device which created *renderPass*. The *renderPass* parameter must be the same as the one given in the VkRenderPassBeginInfo structure for which the render area is relevant. *pGranularity* must point to an instance of the VkExtent2D structure, which will be filled if **vkGetRenderAreaGranularity** is successful. The definitions of the VkExtent2D structure is:

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

The conditions leading to an optimal *renderArea* are:

- the *offset.x* member of *renderArea* must be a multiple of the *width* member of the returned VkExtent2D (the horizontal granularity).
- the *offset.y* member of *renderArea* must be a multiple of the *height* of the returned VkExtent2D (the vertical granularity).
- it must be true that either the *offset.width* member of *renderArea* is a multiple of the horizontal granularity or that *offset.x* + *offset.width* is equal to the *width* of the *framebuffer* in the VkRenderPassBeginInfo.
- it must be true that either the *offset.height* member of *renderArea* is a multiple of the vertical granularity or that *offset.y* + *offset.height* is equal to the *height* of the *framebuffer* in the VkRenderPassBeginInfo.

Valid Usage

- *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*
- Each of *device* and *renderPass* must have been created, allocated or retrieved from the same VkPhysicalDevice

1.119.5 See Also

vkCmdBeginRenderPass

1.120 vkInvalidateMappedMemoryRanges(3)

1.120.1 Name

vkInvalidateMappedMemoryRanges - Invalidate ranges of mapped memory objects.

1.120.2 C Specification

```
VkResult vkInvalidateMappedMemoryRanges(
    VkDevice                                    device,
    uint32_t                                    memoryRangeCount,
    const VkMappedMemoryRange*                  pMemoryRanges);
```

1.120.3 Parameters

device
A handle to the logical device which owns the specified memory ranges.

memoryRangeCount
Number of memory ranges described by *pMemoryRanges*.

pMemoryRanges
Memory ranges to invalidate.

1.120.4 Description

vkInvalidateMappedMemoryRanges invalidates a number of ranges of a number of mapped memory objects. *device* is the handle to the device that owns the memory objects referenced by the call. *memoryRangeCount* specifies the number of memory ranges to invalidate and *pMemoryRanges* is a pointer to an array of VkMappedMemoryRange structures describing the memory ranges to be invalidated. The definition of VkMappedMemoryRange is:

```
typedef struct VkMappedMemoryRange {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceMemory     memory;
    VkDeviceSize       offset;
    VkDeviceSize       size;
} VkMappedMemoryRange;
```

The *sType* member of each element of *pMemoryRanges* should be set to VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE. The *memory* member of the structure specifies the handle to the device memory object containing the mapped region and *offset* and *size* specify the starting offset and size of the region, in bytes, respectively. Areas of regions that extend outside the mapped portion of the parent memory object are ignored and have no effect.

After invalidation, any data stored in the referenced region is discarded and should be considered stale.

Valid Usage

- *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

1.120.5 See Also

vkFlushMappedMemoryRanges

1.121 vkMapMemory(3)

1.121.1 Name

vkMapMemory - Map a memory object into application address space.

1.121.2 C Specification

```
VkResult vkMapMemory(
    VkDevice                                    device,
    VkDeviceMemory                              memory,
    VkDeviceSize                                offset,
    VkDeviceSize                                size,
    VkMemoryMapFlags                            flags,
    void**                                      ppData);
```

1.121.3 Parameters

device
: Logical device which owns the memory object.

memory
: A handle to the memory object to map.

offset
: Start offset of the memory region to map.

size
: Size of the memory region to map.

flags
: This parameter is reserved and must be zero.

ppData
: The pointer to a variable to receive the resulting application-visible address.

1.121.4 Description

`vkMapMemory` maps a region of the memory object specified in `mem` into application address space and returns the resulting pointer in the variable pointed to by `ppData`. The mapped memory region starts at offset *`offset`* and has a size of *`size`*. The `flags` parameter is reserved and should be set to zero.

Valid Usage

- *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*
- Each of *device* and *memory* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *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

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

1.121.5 See Also

vkUnmapMemory, vkFlushMappedMemoryRanges, vkInvalidateMappedMemoryRanges

1.122 vkMergePipelineCaches(3)

1.122.1 Name

vkMergePipelineCaches - Combine the data stores of pipeline caches.

1.122.2 C Specification

```
VkResult vkMergePipelineCaches(
    VkDevice                                    device,
    VkPipelineCache                             dstCache,
    uint32_t                                    srcCacheCount,
    const VkPipelineCache*                      pSrcCaches);
```

1.122.3 Parameters

device
A handle to the device that is the parent of the pipeline caches.

dstCache
The pipeline cache the combined data will be stored into.

srcCacheCount
The number of pipeline caches in the pSrcCaches array.

pSrcCaches
An array of pipeline caches to be combined.

1.122.4 Description

This command combines the caches in the *pSrcCaches* array, storing the result in *dstCache*.

Valid Usage

- *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*
- Each of *device*, *dstCache* and the elements of *pSrcCaches* must have been created, allocated or retrieved from the same `VkPhysicalDevice`
- *dstCache* must not appear in the list of source caches

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

1.122.5 See Also

vkCreatePipelineCache, vkCreateGraphicsPipelines, vkCreateComputePipelines, vkGetPipelineCacheData

1.123 vkQueueBindSparse(3)

1.123.1 Name

vkQueueBindSparse - Bind device memory to a sparse resource object.

1.123.2 C Specification

```
VkResult vkQueueBindSparse(
    VkQueue                                     queue,
    uint32_t                                    bindInfoCount,
    const VkBindSparseInfo*                     pBindInfo,
    VkFence                                     fence);
```

1.123.3 Parameters

queue
: The queue upon which to perform the operation.

bindInfoCount
: The number of binding operations to perform.

pBindInfo
: A pointer to an array of *bindInfoCount* data structures describing the binding operations to perform.

fence
: A handle to a fence object that will be signaled when the binding operation completes.

1.123.4 Description

vkQueueBindSparse binds memory to sparse resources. The number of binding operations to perform is specified in *bindInfoCount* and the binding operation takes place on the queue specified in *queue*. *queue* must be the handle to a queue that is a member of a family that supports the VK_QUEUE_SPARSE_MEMMGR_BIT capability.

pBindInfo is a pointer to an array of *bindInfoCount* VkBindSparseInfo structures describing each of the binding operations. The definition of VkBindSparseInfo is:

```
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;
```

The *sType* member of VkBindSparseInfo should be VK_STRUCTURE_TYPE_BIND_SPARSE_INFO. The *pNext* member is reserved for use by extensions and should be set to **NULL**.

The *waitSemaphoreCount* member specifies the number of semaphores that should be waited on before the binding operation takes place. *pWaitSemaphores* is a pointer to an array of VkSemaphore objects to wait on. If *waitSemaphoreCount* is zero then the value of *pWaitSemaphores* is ignored and the command will not wait before performing the bind operations.

The `bufferBindCount` parameter specifies the number of binding operations to apply to buffer objects. The `pBufferBinds` parameter is a pointer to an array of `bufferBindCount` `VkSparseBufferMemoryBindInfo` structures describing the binding operations for buffers to be performed by the command. The definition of `VkSparseBufferMemoryBindInfo` is:

```
typedef struct VkSparseBufferMemoryBindInfo {
    VkBuffer                    buffer;
    uint32_t                    bindCount;
    const VkSparseMemoryBind*   pBinds;
} VkSparseBufferMemoryBindInfo;
```

The `buffer` member of `VkSparseBufferMemoryBindInfo` specifies the target buffer, `bindCount` specifies the number of binding operations to apply to that buffer, and `pBinds` is a pointer to an array of `VkSparseMemoryBind` structures describing the bindings. If `bindCount` is zero then the value of `pBinds` is ignored and no bindings are performed. The definition of `VkSparseMemoryBind` is:

```
typedef struct VkSparseMemoryBind {
    VkDeviceSize               resourceOffset;
    VkDeviceSize               size;
    VkDeviceMemory             memory;
    VkDeviceSize               memoryOffset;
    VkSparseMemoryBindFlags    flags;
} VkSparseMemoryBind;
```

For each element of the array of `VkSparseMemoryBind` structures, `resourceOffset` and `size` specify the starting offset and size of the region in the buffer. Both are specified in bytes. `memoryOffset` specifies the offset of the region of the memory object specified by `memory` that is to be bound to the specified region in the buffer object. `memoryOffset` is also specified in bytes. `resourceOffset`, `size` and `memoryOffset` must each satisfy the alignment requirements of the buffer. This is returned in the `alignment` field of the `VkMemoryRequirements` structure filled by a call to `vkGetBufferMemoryRequirements`.

The `flags` member of the `VkSparseMemoryBind` structure is a bitfield comprising members of the `VkSparseMemoryBindFlagBits` enumeration, the definition of which is:

```
typedef enum VkSparseMemoryBindFlagBits {
    VK_SPARSE_MEMORY_BIND_METADATA_BIT = 0x00000001,
} VkSparseMemoryBindFlagBits;
```

All other bits in `flags` are reserved and should be set to zero.

The `imageOpaqueBindCount` member specifies the number of opaque image memory binding operations to execute and `pImageOpaqueBinds` is a pointer to an array of `imageOpaqueBindCount` `VkSparseImageOpaqueMemoryBindInfo` structures describing those operations. If `imageOpaqueBindCount` is zero then the value of `pImageOpaqueBinds` is ignored and no binding operations are performed. The definition of `VkSparseImageOpaqueMemoryBindInfo` is:

```
typedef struct VkSparseImageOpaqueMemoryBindInfo {
    VkImage                     image;
    uint32_t                    bindCount;
    const VkSparseMemoryBind*   pBinds;
} VkSparseImageOpaqueMemoryBindInfo;
```

The `image` member of the `VkSparseImageOpaqueMemoryBindInfo` structure specifies the image that is the target of the binding operation. `bindCount` specifies the number of binding operations to be applied to `image` and `pBinds` is a pointer to an array of `VkSparseMemoryBind` structures describing those operations. If `bindCount` is zero then the value of `pBinds` is ignored. Opaque memory binding operations are expressed in terms of byte offsets. The `VkSparseMemoryBind` structures pointed to by the `pBinds` member of `VkSparseImageOpaqueMemoryBindInfo` are interpreted as described for buffers above.

The `imageBindCount` member of the `VkBindSparseInfo` specifies the number of non-opaque image bindings to perform, and the `pImageBinds` member of the structure is a pointer to an array of `imageBindCount` `VkSparseImageMemoryBindInfo` structures describing those operations. If `imageBindCount` is zero then the value of `pImageBinds` is ignored. The definition of `VkSparseImageMemoryBindInfo` is:

```
typedef struct VkSparseImageMemoryBindInfo {
    VkImage                            image;
    uint32_t                           bindCount;
    const VkSparseImageMemoryBind*     pBinds;
} VkSparseImageMemoryBindInfo;
```

Within `VkSparseImageMemoryBindInfo`, the *image* member specifies the image that is to be the target of the binding operation. *bindCount* specifies the number of binding operations to execute and *pBinds* is a pointer to an array of *bindCount* `VkSparseImageMemoryBind` structes describing those operations. The definition of `VkSparseImageMemoryBind` is:

```
typedef struct VkSparseImageMemoryBind {
    VkImageSubresource         subresource;
    VkOffset3D                 offset;
    VkExtent3D                 extent;
    VkDeviceMemory             memory;
    VkDeviceSize               memoryOffset;
    VkSparseMemoryBindFlags    flags;
} VkSparseImageMemoryBind;
```

The *subresource* member of `VkSparseImageMemoryBind` specifies the image subresource within the image to bind memory to. The *offset* and *extent* members specify the region within the image subresource to bind memory to. *offset* and *extent* are instances of the `VkOffset3D` and `VkExtent3D` structures, respectively, and are expressed in texels. The *memoryOffset* member specifies the offset within the memory object specified by *memory* from which to bind memory. *memoryOffset* is expressed in bytes. Its value and the values contained in *offset* and *extent* must satisfy device-specific alignment requirements.

The *flags* member within `VkSparseImageMemoryBind` has the same interpretation as the similarly named member in `VkSparseMemoryBind`.

The *signalSemaphoreCount* parameter specifies the number of semaphores to signal after the binding operations are complete. *pSignalSemaphores* is a pointer to an array of *signalSemaphoreCount* `VkSemaphore` objects to signal at this point. If *signalSemaphoreCount* is zero then the value of *pSignalSemaphores* is ignored and no semaphores are signaled as a result of the operation.

Valid Usage

- *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
- Each of *queue* and *fence* that are valid handles must have been created, allocated or retrieved from the same `VkDevice`
- *fence* must be unsignaled
- *fence* must not be associated with any other queue command that has not yet completed execution on that queue

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
-	-	SPARSE_BINDING

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

1.123.5 See Also

vkQueueSubmit, vkBindBufferMemory, vkGetBufferMemoryRequirements, vkGetImageMemoryRequirements, vkBindImageMemory

1.124 vkQueueSubmit(3)

1.124.1 Name

vkQueueSubmit - Submits a sequence of semaphores or command buffers to a queue.

1.124.2 C Specification

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

1.124.3 Parameters

queue
The queue to which to submit work.

submitCount
The number of submissions to make.

pSubmits
The address of an array of *submitCount* VkSubmitInfo structures to submit to *queue*.

fence
An optional fence object that will be signaled when all command buffers referenced in *pSubmits* have completed execution.

1.124.4 Description

vkQueueSubmit makes one or more submissions to a the queue specified in *queue*. Each submision is represented by an element of an array of VkSubmitInfo structures, the address of which is specified in *pSubmits*. The length of the array is given by *submitCount*. If *submitCount* is zero then, pSubmits may be **NULL**, in which case, no work is submitted to the queue. The definition of VkSubmitInfo is:

```
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;
```

If the *waitSemaphoreCount* member of VkSubmitInfo is not zero, then *pWaitSemaphores* is a pointer to an array of *waitSemaphoreCount* VkSemaphore handles which will be waited on before any further work is performed by the queue.

After all semaphores specified in *pWaitSemaphores* (if any) have become signaled, the command buffers specified in *pCommandBuffers* are executed and those semaphores are again reset to the unsignaled state. *pCommandBuffers* is a pointer to an array of *commandBufferCount* VkCommandBuffer handles to the command buffers to execute. If *commandBufferCount* is zero then *pCommandBuffers* may be **NULL** and no work is performed on the queue as a result.

After all work specified in `pCommandBuffers` (if any) has completed, the semaphores specified in `pSignalSemaphores` are signaled. `pSignalSemaphores` is a pointer to an array of `signalSemaphoreCount` `VkSemaphore` handles. If `signalSemaphoreCount` is zero then `pSignalSemaphores` may be **NULL**.

`fence` is an optional handle to a fence which, if not **VK_NULL_HANDLE**, is signaled when execution of the all elements of `pCommandBuffers` in `pSubmits` is completed. If `submitCount` is zero, but `fence` is not **NULL**, the fence will still be submitted to the queue and will become signaled when all work previusly submitted to the queue has completed.

An implementation may, at its option, choose to merge the submissions specified in `pSubmits`, but at least one submission is made to the queue and the work represented by `pSubmits` is guaranteed to complete in finite time.

Valid Usage

- `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
- Each of `queue` and `fence` that are valid handles must have been created, allocated or retrieved from the same `VkDevice`
- 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

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
-	-	Any

Return Codes

Success

- `VK_SUCCESS`

Failure

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

1.124.5 See Also

`vkBeginCommandBuffer`, `vkEndCommandBuffer`, `vkCreateSemaphore`

1.125 vkQueueWaitIdle(3)

1.125.1 Name

vkQueueWaitIdle - Wait for a queue to become idle.

1.125.2 C Specification

```
VkResult vkQueueWaitIdle(
    VkQueue                                     queue);
```

1.125.3 Parameters

queue
A handle to the queue that is to become idle.

1.125.4 Description

vkQueueWaitIdle waits for all work submitted to the specified queue to complete and for the queue to become idle. After the queue becomes idle, the following guarantees are made:

- Any command buffers previously submitted to the queue have completed execution.
- Any events set or reset by command buffers submitted to that queue will be in their new state.
- Any semaphores signaled by previous calls to vkQueueSubmit will have reached signaled state.

Valid Usage

- *queue* must be a valid VkQueue handle

Command Properties

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

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

1.125.5 See Also

vkDeviceWaitIdle, vkQueueSubmit

1.126 vkResetCommandBuffer(3)

1.126.1 Name

vkResetCommandBuffer - Reset a command buffer.

1.126.2 C Specification

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

1.126.3 Parameters

commandBuffer
Command buffer to reset.

flags
Flags controlling the behavior of the Reset operation. For more details, see `VkCommandBufferResetFlags`.

1.126.4 Description

`vkResetCommandBuffer` resets the command buffer specified in *`commandBuffer`* to a state where it can begin recording commands, i.e. it can be rebuilt by calling **`vkBeginCommandBuffer`**. Note that it is necessary to reset a command buffer which encountered an error during build before it can be reused.

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.

Valid Usage

- *`commandBuffer`* must be a valid `VkCommandBuffer` handle
- *`flags`* must be a valid combination of `VkCommandBufferResetFlagBits` values
- *`commandBuffer`* must not currently be pending execution
- *`commandBuffer`* must have been allocated from a pool that was created with the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`

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`

1.126.5 See Also

vkBeginCommandBuffer, vkEndCommandBuffer

1.127 vkResetCommandPool(3)

1.127.1 Name

vkResetCommandPool - Reset a command pool.

1.127.2 C Specification

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

1.127.3 Parameters

device
The device the command pool was created from.

commandPool
Command pool to reset.

flags
Flags controlling the behavior of the Reset operation. For more details, see `VkCmdPoolResetFlags`.

1.127.4 Description

vkResetCommandPool resets the command pool specified in *commandPool* to a state that depends on the *flags*. If *flags* includes `VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT`, then the pool's memory is returned to the system. If `VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT` is not used, then the pool's memory is return to an "unallocated" state which command buffers can allocate from.

Resetting a pool implicitly resets all command buffers that were created from it, where resetting the command buffers is treated as if `VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT` were used.

Valid Usage

- *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*
- Each of *device* and *commandPool* must have been created, allocated or retrieved from the same `VkPhysicalDevice`
- All `VkCommandBuffer` objects allocated from *commandPool* must not currently be pending execution

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

1.127.5 See Also

vkCreateCommandPool, vkDestroyCommandPool

1.128 vkResetDescriptorPool(3)

1.128.1 Name

vkResetDescriptorPool - Resets a descriptor pool object.

1.128.2 C Specification

```
VkResult vkResetDescriptorPool(
    VkDevice                                    device,
    VkDescriptorPool                            descriptorPool,
    VkDescriptorPoolResetFlags                  flags);
```

1.128.3 Parameters

device
Handle to the logical device which owns the descriptor pool object.

descriptorPool
Handle to the descriptor pool object which needs to be reset.

1.128.4 Description

vkResetDescriptorPool returns all descriptor sets allocated from *descriptorPool* to the pool. This returns the pool to its initial state. *device* must be the handle to the device that owns the pool.

Valid Usage

- *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*
- Each of *device* and *descriptorPool* must have been created, allocated or retrieved from the same VkPhysicalDevice
- All uses of *descriptorPool* (via any allocated descriptor sets) must have completed execution

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

1.128.5 See Also

vkCreateDescriptorPool, vkAllocateDescriptorSets, vkFreeDescriptorSets

1.129 vkResetEvent(3)

1.129.1 Name

vkResetEvent - Reset an event to non-signaled state.

1.129.2 C Specification

```
VkResult vkResetEvent(
    VkDevice                                    device,
    VkEvent                                     event);
```

1.129.3 Parameters

device
Logical device which owns the event.

event
A handle to the event object to reset.

1.129.4 Description

vkResetEvent resets the event object specified by *event* to the non-signaled state.

Valid Usage

- *device* must be a valid VkDevice handle
- *event* must be a valid VkEvent handle
- *event* must have been created, allocated or retrieved from *device*
- Each of *device* and *event* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *event* must not be waited on by a **vkCmdWaitEvents** command that is currently executing

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

1.129.5 See Also

vkSetEvent, vkCreateEvent

1.130 vkResetFences(3)

1.130.1 Name

vkResetFences - Resets one or more fence objects.

1.130.2 C Specification

```
VkResult vkResetFences(
    VkDevice                                    device,
    uint32_t                                    fenceCount,
    const VkFence*                              pFences);
```

1.130.3 Parameters

device
Logical device which owns the specified fences.

fenceCount
Number of fences specified under *pFences*.

pFences
Fences to reset.

1.130.4 Description

vkResetFences resets the status of the *fenceCount* fences whose handles are passed in the *pFences* array to unsignaled.

Valid Usage

- *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*
- Each of *device* and the elements of *pFences* must have been created, allocated or retrieved from the same VkPhysicalDevice
- Any given element of *pFences* must not currently be associated with any queue command that has not yet completed execution on that queue

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

1.130.5 See Also

vkCreateFence, vkDestroyFence, vkGetFenceStatus, vkWaitForFences

1.131 vkSetEvent(3)

1.131.1 Name

vkSetEvent - Set an event to signaled state.

1.131.2 C Specification

```
VkResult vkSetEvent(
    VkDevice                                    device,
    VkEvent                                     event);
```

1.131.3 Parameters

device
Logical device which owns the event.

event
Handle to the event object to signal.

1.131.4 Description

vkSetEvent sets the event object specified by *event* to signaled state. *event* must be an event object owned by *device*. Subsequent calls to vkGetEventStatus on *event* will return the new status. If any command buffers are currently executing on any queue on *device* and are waiting on the event specified by *event*, then they will be unblocked.

Valid Usage

- *device* must be a valid VkDevice handle
- *event* must be a valid VkEvent handle
- *event* must have been created, allocated or retrieved from *device*
- Each of *device* and *event* must have been created, allocated or retrieved from the same VkPhysicalDevice

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

1.131.5 See Also

vkResetEvent, vkGetEventStatus, vkCmdSetEvent, vkCmdWaitEvents

1.132 vkUnmapMemory(3)

1.132.1 Name

vkUnmapMemory - Unmap a previously mapped memory object.

1.132.2 C Specification

```
void vkUnmapMemory(
    VkDevice                                    device,
    VkDeviceMemory                              memory);
```

1.132.3 Parameters

device
Logical device which owns the memory object.

memory
A handle to the memory object to unmap.

1.132.4 Description

`vkUnmapMemory` unmaps the previously mapped memory object specified by *`memory`*.

Valid Usage

- *`device`* must be a valid `VkDevice` handle
- *`memory`* must be a valid `VkDeviceMemory` handle
- *`memory`* must have been created, allocated or retrieved from *`device`*
- Each of *`device`* and *`memory`* must have been created, allocated or retrieved from the same `VkPhysicalDevice`
- *`memory`* must currently be mapped

Host Synchronization

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

1.132.5 See Also

`vkMapMemory`

1.133 vkUpdateDescriptorSets(3)

1.133.1 Name

vkUpdateDescriptorSets - Update the contents of a descriptor set object.

1.133.2 C Specification

```
void vkUpdateDescriptorSets(
    VkDevice                                    device,
    uint32_t                                    descriptorWriteCount,
    const VkWriteDescriptorSet*                 pDescriptorWrites,
    uint32_t                                    descriptorCopyCount,
    const VkCopyDescriptorSet*                  pDescriptorCopies);
```

1.133.3 Parameters

device
: A handle to the device on which to update descriptor sets.

descriptorWriteCount
: Number of descriptor set write requests.

pDescriptorWrites
: Pointer to an array of *descriptorWriteCount* number of VkWriteDescriptorSet structures each specifying the parameters of a descriptor write request to a descriptor set.

descriptorCopyCount
: Number of descriptor set copy requests.

pDescriptorCopies
: Pointer to an array of *descriptorCopyCount* number of VkCopyDescriptorSet structures each specifying the parameters of a descriptor copy request between two descriptor sets.

1.133.4 Description

vkUpdateDescriptorSets allows performing one or more descriptor set update operations.

There are two types of descriptor set update operations: descriptor write and descriptor copy requests.

Descriptor write requests allow writing descriptor data coming from buffer view, image view, and sampler objects to a range of descriptors within a destination descriptor set. Each descriptor write request is described by an instance of the VkWriteDescriptorSet structure. The definition of this structure is:

```
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;
```

Additionally, the structure contains a pointer to an array of *descriptorCount* data structures that specify the buffer view, image view, and/or sampler objects from where the descriptor data is sourced. The information about each of the descriptor updates is stored in up to three arrays, *pTexelBufferInfo*, which is an array of VkBufferView handles, and *pImageInfo* and *pBufferInfo*, which are arrays of VkDescriptorImageInfo and VkDescriptorBufferInfo structures, respectively. The definitions of these structures are as follows:

```
typedef struct VkDescriptorImageInfo {
    VkSampler        sampler;
    VkImageView      imageView;
    VkImageLayout    imageLayout;
} VkDescriptorImageInfo;
```

```
typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset;
    VkDeviceSize    range;
} VkDescriptorBufferInfo;
```

Which of the *pImageInfo*, *pBufferInfo* and *pTexelBufferView* arrays is used is determined from the value of *descriptorType*, as follows:

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, then the entries in *pBufferInfo* are used to update the descriptors and *pImageInfo* and *pTexelBufferInfo* parameters are ignored. For each entry of *pBufferInfo*, *buffer* specifies the handle of the buffer to bind to the descriptor set, and *offset* and *range* specify the starting offset and size of the range of the buffer to bind, respectively, in bytes.

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.

If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, then the *pTexelBufferView* array are used to update the descriptors, and the *pImageInfo* and *pBufferInfo* parameters are 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 members in *pImageInfo* array are used to update the descriptors and the *pBufferInfo* and *pTexelBufferInfo* members are ignored.

Within each element of the *pImageInfo* array, the *sampler* member is a handle to the sampler to bind and is used for descriptor types VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER. The *imageView* is the image view handle used for descriptor updates of type VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT. The *imageLayout* member specifies the layout of the image and is used with descriptor types VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT.

A single descriptor write request may only update a continuous range of descriptors of the same type though that range may cross binding boundaries. See VkWriteDescriptorSet for more information.

Descriptor copy requests allow copying a range of descriptors between a source and destination descriptor set. Each descriptor copy request is described by an instance of the VkCopyDescriptorSet structure, the definition of which is:

```
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;
```

This structure specifies the source and destination descriptor sets of the copy operation in the *srcSet* and *dstSet* members, respectively. The *srcBinding* and *srcArrayElement* members of the structure specify the first entry in the source descriptor set that should be copied; the *dstBinding* and *dstArrayElement* members specify the first entry in the destination descriptor set where the source descriptors should be copied to, while the *descriptorCount* member specifies the number of descriptors to copy. A single descriptor copy request may only copy between two continuous ranges of descriptors of the same type though both the source and desintation ranges may cross binding boundaries. See `VkCopyDescriptorSet` for more information.

Attempting to update the contents of a descriptor set that is used by any command buffer that is pending execution may result in undefined behavior.

Valid Usage

- *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

1.133.5 See Also

`VkWriteDescriptorSet`, `VkCopyDescriptorSet`, `VkDescriptorBufferInfo`, `VkDescriptorImageInfo`

1.134 vkWaitForFences(3)

1.134.1 Name

vkWaitForFences - Wait for one or more fences to become signaled.

1.134.2 C Specification

```
VkResult vkWaitForFences(
    VkDevice                                    device,
    uint32_t                                    fenceCount,
    const VkFence*                              pFences,
    VkBool32                                    waitAll,
    uint64_t                                    timeout);
```

1.134.3 Parameters

device
The device owning the fences to be waited upon.

fenceCount
The number of fences to wait on.

pFences
The address of an array of fences to wait on.

waitAll
If true, wait for all fences to become signaled. Otherwise, wait for at least one fence to become signaled.

timeout
Timeout, in nanoseconds, to wait for fences to become signaled.

1.134.4 Description

vkWaitForFences waits for one or more fences become signaled. *fenceCount* is the number of fences to wait on and *pFences* is a pointer to an array of *fenceCount* fences. If *waitAll* is **VK_TRUE**, then **vkWaitForFences** waits for all fences in the array to become signaled, otherwise it will return when any fence in the array becomes signaled. If none of the fences are signaled before *timeout* nanoseconds elapses, then **vkWaitForFences** will return without any fence necessarily becoming signaled.

Valid Usage

- *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*
- Each of *device* and the elements of *pFences* must have been created, allocated or retrieved from the same VkPhysicalDevice

Return Codes

Success

- VK_SUCCESS
- VK_TIMEOUT

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

1.134.5 Return Value

Upon successful detection of a signaled fence, **vkWaitForFences** returns VK_SUCCESS. If *timeout* nanoseconds pass before any fence becomes signaled, **vkWaitForFences** returns VK_TIMEOUT. Upon failure, a descriptive error code is returned.

1.134.6 See Also

vkCreateFence, vkDestroyFence, vkResetFences

2 Enumerations

2.1 VkDescriptorType(3)

2.1.1 Name

VkDescriptorType - Specifies the type of a descriptor in a descriptor set.

2.1.2 C Specification

```
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;
```

2.1.3 Constants

VK_DESCRIPTOR_TYPE_SAMPLER
Identifies a sampler descriptor which refers the state of a sampler object.
A descriptor of this type enables shaders to perform filtered sampling of any compatible image resource using the referenced sampler in conjunction with a corresponding sampled image descriptor.

VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
Identifies a combined image sampler descriptor which refers the state of a sampler object and an image view object being in a compatible image layout.
A descriptor of this type enables shaders to perform filtered or unfiltered sampling of the referenced image view using the referenced sampler.
This descriptor type is compatible with image views having one of the following image layouts: `VK_IMAGE_LAYOUT_GENERAL`, `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`.
This descriptor type is compatible with image views of image objects created with the `VK_IMAGE_USAGE_SAMPLED_BIT` usage flag.

VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
Identifies a sampled image descriptor which refers the state of an image view object being in a compatible image layout.
A descriptor of this type enables shaders to perform unfiltered sampling of the referenced image view, or can be used in conjunction with a sampler descriptor to perform filtered sampling of the referenced image view.
This descriptor type is compatible with image views in any of the following layouts: `VK_IMAGE_LAYOUT_GENERAL`, `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`.
This descriptor type is compatible with image views of image objects created with the `VK_IMAGE_USAGE_SAMPLED_BIT` usage flag.

VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
Identifies a storage image descriptor which refers the state of an image view object being in a compatible image layout.
A descriptor of this type enables shaders to perform loads, stores, and atomic operations on the referenced image view.

This descriptor type is compatible with image views in any of the following layouts: VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL. However, stores and atomic operations can only be performed on image views in the VK_IMAGE_LAYOUT_GENERAL layout.
This descriptor type is compatible with image views of image objects created with the VK_IMAGE_USAGE_STORAGE_BIT usage flag.

VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
Identifies a uniform texel buffer descriptor which refers the state of a buffer view object.
A descriptor of this type enables shaders to perform reads of uniform texel data from the referenced buffer view.

VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
Identifies a storage texel buffer descriptor which refers the state of a buffer view object.
A descriptor of this type enables shaders to perform loads, stores, and atomic operations on the texel data of the referenced buffer view.

VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
Identifies a uniform buffer descriptor which refers the state of a buffer view object.
A descriptor of this type enables shaders to perform reads of uniform block data in the referenced buffer and offset.

VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
Identifies a storage buffer descriptor which refers the state of a buffer view object.
A descriptor of this type enables shaders to perform loads, stores, and atomic operations of storage block data in the referenced buffer and offset.

VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
Identifies a uniform buffer descriptor with dynamic offset support.
The only difference compared to VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER is that descriptors of this type do not take their offset parameter from the buffer view's corresponding state, but instead allow the application to specify the offset value dynamically at the time the descriptor set containing the descriptor is bound using vkCmdBindDescriptorSets.

VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
Identifies a storage buffer descriptor with dynamic offset support.
The only difference compared to VK_DESCRIPTOR_TYPE_STORAGE_BUFFER is that descriptors of this type do not take their offset parameter from the buffer view's corresponding state, but instead allow the application to specify the offset value dynamically at the time the descriptor set containing the descriptor is bound using vkCmdBindDescriptorSets.

VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
Identifies an input attachment descriptor which refers to the state of an attachment view object.
A descriptor of this type enables shaders to perform loads from images on the referenced attachment view.
This descriptor type is compatible with image views in any of the following layouts: VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL. This descriptor type is compatible with image views of image objects created with the VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT usage flag.

2.1.4 Description

The constants of this enumeration are used to identify the type of the descriptors in various descriptor set and descriptor pool handling commands.

2.1.5 See Also

VkDescriptorSetLayoutBinding

2.2 VkImageLayout(3)

2.2.1 Name

VkImageLayout - Specifies what layout an image object (or a sub-range of it) is in.

2.2.2 C Specification

```
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,
} VkImageLayout;
```

2.2.3 Constants

VK_IMAGE_LAYOUT_UNDEFINED
The contents of images (or a sub-range of it) in this layout are undefined.
This is the layout all images are assumed to be in right after creation, or when their memory binding is changed.
Any operation performed on an image sub-range in this layout leaves the contents of it undefined. Applications need to transition an image sub-range to another layout before being able to perform any operations on it that should result in defined contents.

VK_IMAGE_LAYOUT_GENERAL
An image (or a sub-range of it) in this layout allows all operations to be performed on the image sub-range that is otherwise permitted by the usage flags the image object was created with (see `VkImageUsageFlags` for more detail).

VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
An image (or a sub-range of it) in this layout can only be used as a framebuffer color attachment and as such can only be accessed through framebuffer color reads and writes resulting from the issuing of draw commands, **`vkCmdClearAttachments`**, and through clearing writes resulting from the use of the `VK_ATTACHMENT_LOAD_OP_CLEAR` framebuffer attachment load operation.

VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL
An image (or a sub-range of it) in this layout can only be used as a framebuffer depth/stencil attachment and as such can only be accessed through framebuffer depth/stencil reads and writes resulting from the issuing of draw commands, **`vkCmdClearAttachments`**, and through clearing writes resulting from the use of the `VK_ATTACHMENT_LOAD_OP_CLEAR` framebuffer attachment load operation.

VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL
An image (or a sub-range of it) in this layout can only be used as a read-only framebuffer depth/stencil attachment and as such can only be accessed through framebuffer depth/stencil reads resulting from the issuing of draw commands, and through shader reads done via a sampled image descriptor, combined image sampler descriptor, or read-only storage image descriptor (see `VkDescriptorType` for more detail).

VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
An image (or a sub-range of it) in this layout can only be used as a read-only shader resource and as such can only be accessed by shader reads done via a sampled image descriptor, combined image sampler descriptor, or read-only storage image descriptor (see `VkDescriptorType` for more detail).

VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
An image (or a sub-range of it) in this layout can only be used as the source operand of the commands `vkCmdCopyImage`, `vkCmdBlitImage`, `vkCmdCopyImageToBuffer`, and `vkCmdResolveImage`.

VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL
An image (or a sub-range of it) in this layout can only be used as the destination operand of the commands `vkCmdCopyImage`, `vkCmdBlitImage`, `vkCmdCopyBufferToImage`, `vkCmdResolveImage`, `vkCmdClearColorImage`, and `vkCmdClearDepthStencilImage`.

2.2.4 Description

The constants of this enumeration are used to identify the layout an image object (or a sub-range of it) is expected to be in at any given time, or used to specify the destination layout an image sub-range should be transitioned to as the result of an image memory barrier (see `VkImageMemoryBarrier` for more details).

Performing any operation on an image sub-range that isn't permitted by the layout the image sub-range is currently in is undefined.

If any operation tries to read from an image sub-range that isn't in the expected image layout results in undefined data to be returned as the result of the read.

If any operation tries to write to or perform an atomic operation on an image sub-range that isn't in the expected image layout results in the contents of the whole image to become undefined, i.e. the whole image is logically transitioned to the `VK_IMAGE_LAYOUT_UNDEFINED` layout.

2.2.5 See Also

`VkImageMemoryBarrier`

2.3 VkImageType(3)

2.3.1 Name

VkImageType - Specifies the type of an image object.

2.3.2 C Specification

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

2.3.3 Constants

VK_IMAGE_TYPE_1D
One-dimensional image type.

VK_IMAGE_TYPE_2D
Two-dimensional image type.

VK_IMAGE_TYPE_3D
Three-dimensional image type.

2.3.4 Description

The constants of this enumeration are used to specify the type of an image object created using the `vkCreateImage` command.

2.3.5 See Also

`VkImageCreateInfo`, `vkCreateImage`

2.4 VkImageViewType(3)

2.4.1 Name

VkImageViewType - Specifies the type of an image view object.

2.4.2 C Specification

```
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;
```

2.4.3 Constants

VK_IMAGE_VIEW_TYPE_1D
One-dimensional image view type.
Image views of this type can only be created from image objects of type `VK_IMAGE_TYPE_1D`.

VK_IMAGE_VIEW_TYPE_2D
Two-dimensional image view type.
Image views of this type can only be created from image objects of type `VK_IMAGE_TYPE_2D`.

VK_IMAGE_VIEW_TYPE_3D
Three-dimensional image view type.
Image views of this type can only be created from image objects of type `VK_IMAGE_TYPE_3D`.

VK_IMAGE_VIEW_TYPE_CUBE
Cube image view type.
Image views of this type can only be created from image objects of type `VK_IMAGE_TYPE_2D` that were created using the `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` creation flag.

VK_IMAGE_VIEW_TYPE_1D_ARRAY
One-dimensional array image view type.
Image views of this type can only be created from image objects of type `VK_IMAGE_TYPE_1D`.

VK_IMAGE_VIEW_TYPE_2D_ARRAY
Two-dimensional array image view type.
Image views of this type can only be created from image objects of type `VK_IMAGE_TYPE_2D`.

VK_IMAGE_VIEW_TYPE_CUBE_ARRAY
Cube image array view type.
Image views of this type can only be created from image objects of type `VK_IMAGE_TYPE_2D` that were created using the `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` creation flag.

2.4.4 Description

The constants of this enumeration are used to specify the type of an image view object created using the `vkCreateImageView` command.

2.4.5 See Also

`VkImageViewCreateInfo`, `vkCreateImageView`

2.5 VkSharingMode(3)

2.5.1 Name

VkSharingMode - Specifies the mode of resource sharing.

2.5.2 C Specification

```
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

2.5.3 Constants

VK_SHARING_MODE_EXCLUSIVE
Objects created using this sharing mode can only be accessed by queues of the same queue family at any given time.

VK_SHARING_MODE_CONCURRENT
Objects created using this sharing mode can be accessed by queues from different queue families simultaneously.

2.5.4 Description

The constants of this enumeration are used to specify the intended resource sharing mode used by a buffer or image object.

Buffers and images created using `VK_SHARING_MODE_EXCLUSIVE` can only be accessed by queues of the same queue family at any given time. Before being able to access the object using a queue from a different queue family the application has to transfer exclusive ownership of the object between the source and destination queue families. In order to do that the application has to perform the following operations:

1. Release exclusive ownership from the source queue family to the destination queue family.
2. Use semaphores to ensure proper execution control for the ownership transfer.
3. Acquire exclusive ownership for the destination queue family from the source queue family.

To release exclusive ownership the application should execute an image memory barrier (see `VkImageMemoryBarrier`) on a queue from the source queue family where it must set the *`srcQueueFamilyIndex`* parameter of the barrier to the source queue family's index, and the *`dstQueueFamilyIndex`* parameter of the barrier to the destination queue family's index.

To acquire exclusive ownership the application should execute the same image memory barrier on a queue from the destination queue family.

Buffers and images created using `VK_SHARING_MODE_CONCURRENT` can be simultaneously accessed by queues from different queue families. Accesses of buffers and images created using this sharing mode may have lower performance characteristics.

2.5.5 See Also

`VkImageCreateInfo`, `VkBufferCreateInfo`

3 Flags

3.1 VkBufferCreateFlags(3)

3.1.1 Name

VkBufferCreateFlags - Buffer object creation flags.

3.1.2 C Specification

```
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;
```

```
typedef VkFlags VkBufferCreateFlags;
```

3.1.3 Constants

VK_BUFFER_CREATE_SPARSE_BINDING_BIT
Buffer objects created with this flag allow their contents to backed by sparse memory allocations using `vkQueueBindSparse`.

VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT
Buffer objects created with this flag allow their contents to be backed by a partially resident sparse memory allocation.

VK_BUFFER_CREATE_SPARSE_ALIASED_BIT
Buffer objects created with this flag allow their contents to be backed by a sparse memory allocation that might also simultaneously be backing another buffer (or another portion of the buffer).

3.1.4 Description

These flags are used in the `VkBufferCreateInfo` structure passed as parameter to `vkCreateBuffer` to define the properties of the created buffer object.

3.1.5 See Also

`VkBufferCreateInfo`, `vkCreateBuffer`

3.2 VkBufferUsageFlags(3)

3.2.1 Name

VkBufferUsageFlags - Buffer object usage flags.

3.2.2 C Specification

```
typedef VkFlags VkBufferUsageFlags;
```

3.2.3 Constants

VK_BUFFER_USAGE_TRANSFER_SRC_BIT
The buffer can be used as the source operand of transfer operations (`vkCmdCopyBuffer`, `vkCmdCopyBufferToImage`).

VK_BUFFER_USAGE_TRANSFER_DST_BIT
The buffer can be used as the destination operand of transfer operations (`vkCmdCopyBuffer`, `vkCmdCopyImageToBuffer`, `vkCmdUpdateBuffer`, `vkCmdFillBuffer`, `vkCmdWriteTimestamp`, `vkCmdCopyQueryPoolResults`).

VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT
The buffer supports reads via uniform texel buffer descriptors.

VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT
The buffer supports loads, stores, and atomic operations via storage texel buffer descriptors.

VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT
The buffer supports reads via uniform buffer descriptors.

VK_BUFFER_USAGE_STORAGE_BUFFER_BIT
The buffer supports loads, stores, and atomic operations via storage buffer descriptors.

VK_BUFFER_USAGE_INDEX_BUFFER_BIT
The buffer can be bound as an index buffer using the `vkCmdBindIndexBuffer` command.

VK_BUFFER_USAGE_VERTEX_BUFFER_BIT
The buffer can be bound as a vertex buffer using the `vkCmdBindVertexBuffers` command.

VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT
The buffer can be used as the source of indirect commands (`vkCmdDrawIndirect`, `vkCmdDrawIndexedIndirect`, `vkCmdDispatchIndirect`).

3.2.4 Description

These flags are used in the `VkBufferCreateInfo` structure passed as parameter to `vkCreateBuffer` to define the intended use of the created buffer. Trying to use the buffer for any other purpose than those requested at creation time may result in undefined behavior.

3.2.5 See Also

`VkBufferCreateInfo`, `vkCreateBuffer`

3.3 VkFormatFeatureFlags(3)

3.3.1 Name

VkFormatFeatureFlags - Capability flags of a particular format.

3.3.2 C Specification

```
typedef VkFlags VkFormatFeatureFlags;
```

3.3.3 Constants

VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT
Image views having this format support filtered and/or unfiltered sampling via sampled image and combined image sampler descriptors.

VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT
Image views having this format support loads and stores via storage image descriptors.

VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT
Image views having this format support atomic operations via storage image descriptors.

VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT
Buffer views having this format support uniform reads via uniform texel buffer descriptors.

VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT
Buffer views having this format support loads and stores via storage texel buffer descriptors.

VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT
Buffer views having this format support atomic operations via storage texel buffer descriptors.

VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT
Indicates that the format is supported for vertex attributes.

VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT
Indicates that the format is supported for color attachment views and thus can be used as framebuffer color attachment format.

VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT
Framebuffer color attachments having this format also support blending.
If this flag is present then `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` is also present.

VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
Indicates that the format is supported for depth/stencil views and thus can be used as framebuffer depth/stencil attachment format.

VK_FORMAT_FEATURE_BLIT_SRC_BIT
Format can be used as the source image of blits with `vkCmdBlitImage`

VK_FORMAT_FEATURE_BLIT_DST_BIT
Format can be used as the destination image of blits with `vkCmdBlitImage`

3.3.4 Description

These flags are used in the `VkFormatProperties` structure that is returned by `vkGetPhysicalDeviceFormatProperties`.

3.3.5 See Also

`VkFormatProperties`, `vkGetPhysicalDeviceFormatProperties`

3.4 VkImageCreateFlags(3)

3.4.1 Name

VkImageCreateFlags - Image object creation flags.

3.4.2 C Specification

```
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,
} VkImageCreateFlagBits;
```

```
typedef VkFlags VkImageCreateFlags;
```

3.4.3 Constants

VK_IMAGE_CREATE_SPARSE_BINDING_BIT
Image objects created with this flag allow their contents to backed by sparse memory allocations using `vkQueueBindSparse`.

VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
Image objects created with this flag allow their contents to be backed by a partially resident sparse memory allocation.

VK_IMAGE_CREATE_SPARSE_ALIASED_BIT
Image objects created with this flag allow their contents to be backed by a sparse memory allocation that might also simultaneously be backing another image (or another portion of the image).

VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT
Image objects created with this flag allow image view objects created from them to override the format of the image to any compatible format. Otherwise image view objects created from the image must match the format of the image object.

VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT
Image objects created with this flag allow image view objects of type `VK_IMAGE_VIEW_TYPE_CUBE` to be created from. This flag is only allowed to be used if the image object's type is `VK_IMAGE_TYPE_2D`.

3.4.4 Description

These flags are used in the `VkImageCreateInfo` structure passed as parameter to `vkCreateImage` to define the properties of the created image object.

3.4.5 See Also

`VkImageCreateInfo`, `vkCreateImage`

3.5 VkImageUsageFlags(3)

3.5.1 Name

VkImageUsageFlags - Image object usage flags.

3.5.2 C Specification

```
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;
```

```
typedef VkFlags VkImageUsageFlags;
```

3.5.3 Constants

VK_IMAGE_USAGE_TRANSFER_SRC_BIT
The image can be used as the source operand of transfer operations (vkCmdCopyImage, vkCmdBlitImage, vkCmdCopyImageToBuffer, vkCmdResolveImage).

VK_IMAGE_USAGE_TRANSFER_DST_BIT
The image can be used as the destination operand of transfer operations (vkCmdCopyImage, vkCmdBlitImage, vkCmdCopyBufferToImage, vkCmdClearColorImage, vkCmdClearDepthStencilImage, vkCmdResolveImage).

VK_IMAGE_USAGE_SAMPLED_BIT
The image supports filtered and/or unfiltered sampling via sampled image and combined image sampler descriptors.

VK_IMAGE_USAGE_STORAGE_BIT
The image supports loads, stores, and atomic operations via storage image descriptors.

VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
The image can be used as a framebuffer color attachment.
A framebuffer can only use an attachment view as a color attachment if the view's image was created with this usage flag.

VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
The image can be used as a framebuffer depth/stencil attachment.
A framebuffer can only use an attachment view as a depth/stencil attachment if the view's image was created with this usage flag.

VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
The contents of images created with this usage flag are only maintained within a render pass.

VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
The image can be used as a framebuffer input attachment.

3.5.4 Description

These flags are used in the VkImageCreateInfo structure passed as parameter to vkCreateImage to define the intended use of the created image. Trying to use the image for any other purpose than those requested at creation time may result in undefined behavior.

VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT and VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT are exclusive. No image can be created with both of these flags being set.

VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT can only be used together with either the VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, the VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, or the VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT flags.

3.5.5 See Also

VkImageCreateInfo, vkCreateImage

3.6 VkMemoryPropertyFlags(3)

3.6.1 Name

VkMemoryPropertyFlags - Memory pool properties.

3.6.2 C Specification

```
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;
```

```
typedef VkFlags VkMemoryPropertyFlags;
```

3.6.3 Constants

VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
Identifies a memory pool that is the most efficient for device access.

VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
Identifies a memory pool that can be mapped into host memory address space and thus is accessible by the host.

VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
Identifies a memory pool where accesses between the host and the coherency domain are coherent. Memory without this property requires explicit use of `vkFlushMappedMemoryRanges` after host writes to this type of memory, and use of `vkInvalidateMappedMemoryRanges` before host reads from that memory.

VK_MEMORY_PROPERTY_HOST_CACHED_BIT
Identifies memory that is cached by the host.

VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT
Identifies memory where an object's backing may be provided lazily (when needed) by the implementation.

3.6.4 Description

These flags are used in the `VkMemoryAllocateInfo` structure passed as parameter to `vkAllocateMemory` to define the properties of the memory pool the memory object should be allocated from.

Additionally, when querying the memory requirements of objects using `vkGetBufferMemoryRequirements` or `vkGetImageMemoryRequirements`, the *`memoryTypeBits`* member returned in the `VkMemoryRequirements` structure takes its values from these set of flags.

3.6.5 See Also

`VkMemoryAllocateInfo`, `vkAllocateMemory`, `VkMemoryRequirements`

3.7 VkPipelineStageFlags(3)

3.7.1 Name

VkPipelineStageFlags - Pipeline stage identifiers.

3.7.2 C Specification

```
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,
} VkPipelineStageFlagBits;
```

3.7.3 Constants

VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
Stage of the pipeline where commands are initially received by the queue.

VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
Stage of the pipeline where Draw/DispatchIndirect data structures are consumed.

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/stencil test before fragment shading) are performed.

VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
Stage of the pipeline where late fragment tests (depth/stencil test after fragment shading) are performed.

VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
Stage of the pipeline after blending where the final color values are output from the pipeline. Note that this does not necessarily indicate that the values have been committed to memory.

VK_PIPELINE_STAGE_TRANSFER_BIT
Execution of copy commands. This includes the operations resulting from all transfer commands, e.g. **vkCmdCopyImage**, **vkCmdCopyBuffer**, **vkCmdBlitImage**, **vkCmdResolveImage**, **vkCmdClearColorImage**, etc.

VK_PIPELINE_STAGE_HOST_BIT
Indicates execution on the Host of reads/writes of device memory.

3.7.4 Description

The pipeline stages are used to describe which operations must be synchronized for the purposes of execution dependencies in pipeline barriers, event signal/wait, and subpass dependencies. Many of the bits describe stages of the graphics pipeline, but there are also pseudo-stages for compute work, copy commands, and CPU production/consumption of data.

For the commands vkCmdSetEvent and vkCmdResetEvent the event object is set and reset, respectively, after the specified pipeline stages have completed executing prior commands. Some implementations may not be able to signal at as fine a grain as the bits in the bitfield, in which case the signaling may occur after additional stages have completed executing prior commands.

For the command vkCmdWaitEvents, the *srcStageMask* should be a bitwise OR of all *stageMasks* used to signal the events, as described above. If some events were signaled with vkSetEvent, then this should include VK_PIPELINE_STAGE_HOST_BIT. *dstStageMask* indicates the set of pipeline stages that should not begin executing subsequent commands until the events are signaled. Some implementations may not be able to wait at as fine a grain as the bits in the bitfield, in which case the waiting may occur at an earlier stage in the pipeline.

For the command vkCmdPipelineBarrier, writes as described by *outputMask* that were written by pipeline stages in *srcStageMask* prior to the barrier are made visible to reads as described by *inputMask* in pipeline stages in *dstStageMask* subsequent to the barrier.

VK_PIPELINE_STAGE_HOST_BIT cannot be used for **vkCmdSetEvent**, **vkCmdResetEvent**, or **vkCmdPipelineBarrier**.

3.7.5 See Also

vkCmdSetEvent, vkCmdResetEvent, vkCmdWaitEvents, vkCmdPipelineBarrier

3.8 VkQueryControlFlags(3)

3.8.1 Name

VkQueryControlFlags - Query control flags.

3.8.2 C Specification

```
typedef VkFlags VkQueryControlFlags;
```

3.8.3 Constants

VK_QUERY_CONTROL_PRECISE_BIT
When this flag is used the query must collect precise results. Without this flag the actual result of occlusion queries may be less than the result of the same query when using this flag.

3.8.4 Description

These flags are used to control the behavior of queries started with the `vkCmdBeginQuery` command.

3.8.5 See Also

`vkCmdBeginQuery`

3.9 VkQueryResultFlags(3)

3.9.1 Name

VkQueryResultFlags - Query result flags.

3.9.2 C Specification

```
typedef VkFlags VkQueryResultFlags;
```

3.9.3 Constants

VK_QUERY_RESULT_32_BIT
When this flag is used the results of the queries are written to the destination buffer as one or more 32-bit values.

VK_QUERY_RESULT_64_BIT
When this flag is used the results of the queries are written to the destination buffer as one or more 64-bit values.

VK_QUERY_RESULT_NO_WAIT_BIT
When this flag is used the results of the queries aren't waited on before proceeding with the result copy.

VK_QUERY_RESULT_WAIT_BIT
When this flag is used the results of the queries are waited on before proceeding with the result copy.

VK_QUERY_RESULT_WITH_AVAILABILITY_BIT
When this flag is used the availability of the results is also written to the destination buffer as a separate value after the actual results. If the results of the query were available at the time of the result copy the integer value 1 is written, otherwise the integer value 0 is written to the destination buffer.

VK_QUERY_RESULT_PARTIAL_BIT
When this flag is used the partial results of the queries are written to the destination buffer even if the final results aren't available. If this flag isn't used then the locations in the destination buffer corresponding to result values of queries whose result isn't available at the time of the result copy will be left untouched.

3.9.4 Description

These flags are used to control the behavior of the query result copy commands `vkGetQueryPoolResults` and `vkCmdCopyQueryPoolResults`.

3.9.5 See Also

`vkGetQueryPoolResults`, `vkCmdCopyQueryPoolResults`

3.10 VkQueueFlags(3)

3.10.1 Name

VkQueueFlags - Queue capability flags.

3.10.2 C Specification

```
typedef VkFlags VkQueueFlags;
```

3.10.3 Constants

VK_QUEUE_GRAPHICS_BIT
Queues which have this capability flag support graphics operations. These operations include support using a graphics pipeline and issuing draw commands.

VK_QUEUE_COMPUTE_BIT
Queues which have this capability flag support compute operations. These operations include support using a compute pipeline and issuing dispatch commands.

VK_QUEUE_TRANSFER_BIT
Queues which have this capability flag support transfer operations. These operations include all of the copy commands.

VK_QUEUE_SPARSE_BINDING_BIT
Queues which have this capability flag support memory management operations. These operations are affected by calling `vkQueueBindSparse`.

3.10.4 Description

These flags are returned in the `VkQueueFamilyProperties` structure together with other capabilities of a queue from a particular queue family as result of calling `vkGetPhysicalDeviceQueueFamilyProperties`.

3.10.5 See Also

`VkQueueFamilyProperties`, `vkGetPhysicalDeviceQueueFamilyProperties`

4 Structures

4.1 VkBufferCreateInfo(3)

4.1.1 Name

VkBufferCreateInfo - Structure specifying the parameters of a newly created buffer object.

4.1.2 C Specification

```
typedef struct VkBufferCreateInfo {
    VkStructureType        sType;
    const void*            pNext;
    VkBufferCreateFlags    flags;
    VkDeviceSize           size;
    VkBufferUsageFlags     usage;
    VkSharingMode          sharingMode;
    uint32_t               queueFamilyIndexCount;
    const uint32_t*        pQueueFamilyIndices;
} VkBufferCreateInfo;
```

4.1.3 Fields

sType
: Structure type. Must be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`.

pNext
: Pointer to next structure in the structure chain when applicable.

size
: Size of the buffer in bytes.

usage
: Allowed usages of the buffer (see `VkBufferUsageFlags` for more detail).

flags
: Other properties of the buffer (see `VkBufferCreateFlags` for more detail).

sharingMode
: Sharing mode used for the buffer (see `VkSharingMode` for more detail).

queueFamilyIndexCount
: Number of queue families that can access the buffer in case *sharingMode* is `VK_SHARING_MODE_CONCURRENT`.

pQueueFamilyIndices
: Array of *queueFamilyIndexCount* queue family indices specifying the set of queue families that can access the buffer in case *sharingMode* is `VK_SHARING_MODE_CONCURRENT`.

4.1.4 Description

This structure is used to specify the parameters of buffer objects created using `vkCreateBuffer`.

Valid Usage

- *sType* must be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`
- *pNext* must be `NULL`
- *flags* must be a valid combination of `VkBufferCreateFlagBits` values
- *usage* must be a valid combination of `VkBufferUsageFlagBits` values
- *usage* must not be `0`
- *sharingMode* must be a valid `VkSharingMode` value
- *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`

4.1.5 See Also

`vkCreateBuffer`

4.2 VkBufferMemoryBarrier(3)

4.2.1 Name

VkBufferMemoryBarrier - Structure specifying the parameters of a buffer memory barrier.

4.2.2 C Specification

```
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;
```

4.2.3 Fields

sType
Structure type. Must be `VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER`.

pNext
Pointer to next structure in the structure chain when applicable.

outputMask
Types of writes to the buffer to flush (see `VkMemoryOutputFlags` for more detail).

inputMask
Types of reads from the buffer to invalidate (see `VkMemoryInputFlags` for more detail).

srcQueueFamilyIndex
Identifies the source queue family to transfer ownership of the buffer from. A value of `VK_QUEUE_FAMILY_IGNORED` indicates that this member should be ignored.

destQueueFamilyIndex
Identifies the destination queue family to transfer ownership of the buffer to. A value of `VK_QUEUE_FAMILY_IGNORED` indicates that this member should be ignored.

buffer
Buffer object the memory barrier applies to.

offset
Byte offset of the sub-range of the buffer the memory barrier applies to.

size
Size in bytes of the sub-range of the buffer the memory barrier applies to.

Valid Usage

- *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
- *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, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must both be VK_QUEUE_FAMILY_IGNORED
- If *buffer* was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must either both be VK_QUEUE_FAMILY_IGNORED, or both be a valid queue family (see [?])
- 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

4.2.4 Description

This structure specifies the parameters of a buffer memory barrier that can be passed in the *ppMemoryBarriers* parameter of vkCmdPipelineBarrier and vkCmdWaitEvents.

4.2.5 See Also

vkCmdPipelineBarrier, vkCmdWaitEvents, VkMemoryBarrier, VkImageMemoryBarrier

4.3 VkDescriptorSetAllocateInfo(3)

4.3.1 Name

VkDescriptorSetAllocateInfo - Structure specifying the allocation parameters for descriptor sets.

4.3.2 C Specification

```
typedef struct VkDescriptorSetAllocateInfo {
    VkStructureType                  sType;
    const void*                      pNext;
    VkDescriptorPool                 descriptorPool;
    uint32_t                         descriptorSetCount;
    const VkDescriptorSetLayout*     pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

4.3.3 Fields

sType
: Structure type. Must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO`.

pNext
: Pointer to next structure in the structure chain when applicable.

descriptorPool
: The pool from which to allocate the descriptor sets.

descriptorSetCount
: The number of descriptor sets to allocate.

pSetLayouts
: An array of *descriptorSetCount* handles to descriptor set layouts objects describing the descriptor sets.

4.3.4 Description

This structure is used to specify the parameters of descriptor set objects allocated using `vkAllocateDescriptorSets`.

Valid Usage

- *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`
- Each of *descriptorPool* and the elements of *pSetLayouts* must have been created, allocated or retrieved from the same `VkDevice`
- *descriptorSetCount* must not be greater than the number of sets that are currently available for allocation in *descriptorPool*
- *descriptorPool* must have enough free descriptor capacity remaining to allocate the descriptor sets of the specified layouts

4.3.5 See Also

vkAllocateDescriptorSets

4.4 VkImageCreateInfo(3)

4.4.1 Name

VkImageCreateInfo - Structure specifying the parameters of a newly created image object.

4.4.2 C Specification

```
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;
```

4.4.3 Fields

sType
Structure type. Must be `VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO`.

pNext
Pointer to next structure in the structure chain when applicable.

imageType
Type of the image (see `VkImageType` for more detail).

format
Format of the texels of the image (see `VkFormat` for more detail).

extent
Width, height, and depth of the image in texels.

mipLevels
Number of mip levels of the image.

arrayLayers
Number of layers of the image.

samples
Number of samples of the image.

tiling
Image tiling mode of the image (see `VkImageTiling` for more detail).

usage
Allowed usages of the image (see `VkImageUsageFlags` for more detail).

flags
Other properties of the image (see `VkImageCreateFlags` for more detail).

sharingMode
Sharing mode used for the image (see `VkSharingMode` for more detail).

queueFamilyIndexCount
Number of queue families that can access the image in case *sharingMode* is VK_SHARING_MODE_CONCURRENT.

pQueueFamilyIndices
Array of *queueFamilyIndexCount* queue family indices specifying the set of queue families that can access the image in case *sharingMode* is VK_SHARING_MODE_CONCURRENT.

4.4.4 Description

This structure is used to specify the parameters of image objects created using `vkCreateImage`.

Valid Usage

- *sType* must be `VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO`
- *pNext* must be `NULL`
- *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 *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`
- The *width*, *height*, and *depth* members of *extent* must all be greater than `0`
- *mipLevels* must be greater than `0`
- *arrayLayers* must be greater than `0`
- 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
- 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 $\lfloor \log_2(\max(extent.width, extent.height, extent.depth)) \rfloor + 1$
- 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

4.4.5 See Also

vkCreateImage

4.5 VkImageMemoryBarrier(3)

4.5.1 Name

VkImageMemoryBarrier - Structure specifying the parameters of an image memory barrier.

4.5.2 C Specification

```
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;
```

4.5.3 Fields

sType
: Structure type. Must be `VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER`.

pNext
: Pointer to next structure in the structure chain when applicable.

outputMask
: Types of writes to the image to flush (see `VkMemoryOutputFlags` for more detail).

inputMask
: Types of reads from the image to invalidate (see `VkMemoryInputFlags` for more detail).

oldLayout
: Current layout the image is expected to be in (see `VkImageLayout` for more detail).

newLayout
: New layout the image should be transferred to (see `VkImageLayout` for more detail).

srcQueueFamilyIndex
: Identifies the source queue family to transfer ownership of the image from. A value of `VK_QUEUE_FAMILY_IGNORED` indicates that this member should be ignored.

dstQueueFamilyIndex
: Identifies the destination queue family to transfer ownership of the image to. A value of `VK_QUEUE_FAMILY_IGNORED` indicates that this member should be ignored.

image
: Image object the memory barrier applies to.

subresourceRange
: Sub-range of the image the memory barrier applies to.

4.5.4 Description

This structure specifies the parameters of an image memory barrier that can be passed in the *ppMemBarriers* parameter of `vkCmdPipelineBarrier` and `vkCmdWaitEvents`.

Valid Usage

- *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
- *oldLayout* must be `VK_IMAGE_LAYOUT_UNDEFINED`, `VK_IMAGE_LAYOUT_PREINITIALIZED` or the current layout of the image region 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`, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must both be `VK_QUEUE_FAMILY_IGNORED`
- If *image* was created with a sharing mode of `VK_SHARING_MODE_EXCLUSIVE`, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must either both be `VK_QUEUE_FAMILY_IGNORED`, or both be a valid queue family (see [?])
- 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 [?])
- 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

4.5.5 See Also

vkCmdPipelineBarrier, vkCmdWaitEvents, VkMemoryBarrier, VkBufferMemoryBarrier

4.6 VkPhysicalDeviceFeatures(3)

4.6.1 Name

VkPhysicalDeviceFeatures - Structure describing the fine-grained features that can be supported by an implementation.

4.6.2 C Specification

```
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;
```

4.6.3 Fields

robustBufferAccess
out of bounds buffer accesses are well defined

fullDrawIndexUint32
full 32-bit range of indices are supported for indexed draw calls using VK_INDEX_TYPE_UINT32.

imageCubeArray
image views which are arrays of cube maps are supported (VK_IMAGE_VIEW_TYPE_CUBE_ARRAY)

independentBlend
blending operations are controlled independently per-attachment

geometryShader
geometry shader stage is supported

tessellationShader
tessellation control and evaluation shader stages are supported

sampleRateShading
per-sample shading and multisample interpolation are supported

dualSrcBlend
blend operations which take two sources are supported

logicOp
logic operations are supported

multiDrawIndirect
multi draw indirect is supported

depthClamp
depth clamping is supported

depthBiasClamp
depth bias clamping is supported

fillModeNonSolid
point and wireframe fill modes are supported

depthBounds
depth bounds test is supported

wideLines
lines with width greater than 1 are supported

largePoints
points with size greater than 1 are supported

alphaToOne
the implementation can replace the alpha value of the color fragment output to the maximum representable alpha value for fixed-point colors or 1.0 for floating-point colors.

multiViewport
mulitple viewports are supported

samplerAnisotropy
Anisotropic filtering is supported

textureCompressionETC2
ETC and EAC texture compression formats are supported

textureCompressionASTC_LDR
ASTC LDR texture compression formats are supported

textureCompressionBC
BC1-7 texture compressed formats are supported

pipelineStatisticsQuery
pipeline statistics queries are supported

vertexPipelineStoresAndAtomics
storage buffers and images support stores and atomic operationscan in the vertex, tessellation, and geometry shader stages.

fragmentStoresAndAtomics
storage buffers and images support stores and atomic operationscan in the fragment shader stage.

shaderTessellationAndGeometryPointSize
the *PointSize* shader builtin is available in the tessellation control, tessellation evaluation, and geometry shader stages.

shaderImageGatherExtended
image gather with *non-constant offset* and image gather with *offsets* are supported

shaderStorageImageExtendedFormats
the extended set of image formats can be used for storage images

shaderStorageImageMultisample
multisample images can be used for storage images

shaderUniformBufferArrayDynamicIndexing
arrays of uniform buffers can be accessed with dynamically uniform indices

shaderSampledImageArrayDynamicIndexing
arrays of samplers and sampled images can be accessed with dynamically uniform indices

shaderStorageBufferArrayDynamicIndexing
arrays of storage buffers can be accessed with dynamically uniform indices

shaderStorageImageArrayDynamicIndexing
arrays of storage images can be accessed with dynamically uniform indices

shaderClipDistance
clip distance is supported in shader code

shaderCullDistance
cull distance is supported in shader code

shaderFloat64
64-bit floats (doubles) are supported in shader code

shaderInt64
64-bit integers are supported in shader code

shaderInt16
16-bit integers are supported in shader code

shaderResourceResidency
image operations that return resource residency information are supported in shader code

shaderResourceMinLod
: image operations that specify minimum resource Lod are supported in shader code

sparseBinding
: indicates whether resource memory can be managed at opaque page level

shaderResourceResidency
: Sparse resources support: Resource memory can be managed at opaque page level rather than object level

sparseResidencyBuffer
: Sparse resources support: physical device can access partially resident buffers

sparseResidencyImage2D
: Sparse resources support: physical device can access partially resident 2D (non-MSAA non-DepthStencil) images

sparseResidencyImage3D
: Sparse resources support: physical device can access partially resident 3D images

sparseResidency2Samples
: Sparse resources support: physical device can access partially resident MSAA 2D images with 2 samples

sparseResidency4Samples
: Sparse resources support: physical device can access partially resident MSAA 2D images with 4 samples

sparseResidency8Samples
: Sparse resources support: physical device can access partially resident MSAA 2D images with 8 samples

sparseResidency16Samples
: Sparse resources support: physical device can access partially resident MSAA 2D images with 16 samples

sparseResidencyAliased
: Sparse resources support: physical device can correctly access data aliased into multiple locations (opt-in)

4.6.4 Description

The VkPhysicalDeviceFeatures structure contains a feature flag for each of the fine-grained features that may be supported by an implementation.

When passed to `vkGetPhysicalDeviceFeatures` as the *`pFeatures`* parameter, the implementation will fill in each member of the structure with `VK_TRUE` if the indicated *`physicalDevice`* supports the feature, or with `VK_FALSE` if the physical device does not support the feature.

Fine-grained features must be enabled at `VkDevice` creation time. This is done by passing a pointer to a `VkPhysicalDeviceFeatures` structure in the *`pEnabledFeatures`* member of the `VkDeviceCreateInfo` structure that is passed to `vkCreateDevice`. In this case, setting a member of the structure to `VK_TRUE` will enable support for the feature on the indictated physical device, and setting a member to `VK_FALSE` will disable support for the feature.

include::../validity/structs/VkPhysicalDeviceFeatures .txt[]

4.6.5 See Also

`vkGetPhysicalDeviceFeatures`, `vkGetPhysicalDeviceProperties`, `vkCreateDevice`

4.7 VkPhysicalDeviceLimits(3)

4.7.1 Name

VkPhysicalDeviceLimits - Structure

4.7.2 C Specification

```
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;
```

4.7.3 Fields

maxImageDimension1D

max 1D image dimension

maxImageDimension2D
max 2D image dimension

maxImageDimension3D
max 3D image dimension

maxImageDimensionCube
max cubemap image dimension

maxImageArrayLayers
max layers for image arrays

maxTexelBufferSize
max texel buffer size (bytes)

maxUniformBufferRange
max uniform buffer range (bytes)

maxStorageBufferRange
max storage buffer range (bytes)

maxPushConstantsSize
max size of the push constants pool (bytes)

maxMemoryAllocationCount
max number of device memory allocations supported

maxSamplerAllocationCount
max number of samplers that can be allocated on a device

bufferImageGranularity
Granularity (in bytes) at which buffers and linear images vs optimal images can be bound to adjacent memory locations without aliasing

maxBoundDescriptorSets
max number of descriptors sets that can be bound to a pipeline

maxPerStageDescriptorSamplers
max num of samplers allowed per-stage in a descriptor set

maxPerStageDescriptorUniformBuffers
max num of uniform buffers allowed per-stage in a descriptor set

maxPerStageDescriptorStorageBuffers
max num of storage buffers allowed per-stage in a descriptor set

maxPerStageDescriptorSampledImages
max num of sampled images allowed per-stage in a descriptor set

maxPerStageDescriptorStorageImages
max num of storage images allowed per-stage in a descriptor set

maxPerStageDescriptorInputAttachments
max num of input attachments allowed per-stage in a descriptor set

maxDescriptorSetUniformBuffers
max num of uniform buffers allowed in all stages in a descriptor set

maxDescriptorSetStorageBuffers
max num of storage buffers allowed in all stages in a descriptor set

maxDescriptorSetSampledImages
max num of sampled images allowed in all stages in a descriptor set

maxDescriptorSetStorageImages
max num of storage images allowed in all stages in a descriptor set

maxDescriptorSetInputAttachments
max num of input attachments allowed in all stages in a descriptor set

maxVertexInputAttributes
max num of vertex input attribute slots maxVertexInputBindings: max num of vertex input binding slots

maxVertexInputAttributeOffset
max vertex input attribute offset added to vertex buffer offset

maxVertexInputBindingStride
max vertex input binding stride

maxVertexOutputComponents
max num of output components written by vertex shader

maxTessellationGenLevel
max level supported by tessellation primitive generator

maxTessellationPatchSize
max patch size (vertices)

maxTessellationControlPerVertexInputComponents
max num of input components per-vertex in TCS

maxTessellationControlPerVertexOutputComponents
max num of output components per-vertex in TCS

maxTessellationControlPerPatchOutputComponents
max num of output components per-patch in TCS

maxTessellationControlTotalOutputComponents
max total num of per-vertex and per-patch output components in TCS

maxTessellationEvaluationInputComponents
max num of input components per vertex in TES

maxTessellationEvaluationOutputComponents
max num of output components per vertex in TES

maxGeometryShaderInvocations
max invocation count supported in geometry shader

maxGeometryInputComponents
max num of input components read in geometry stage

maxGeometryOutputComponents
max num of output components written in geometry stage

maxGeometryOutputVertices
max num of vertices that can be emitted in geometry stage

maxGeometryTotalOutputComponents
max total num of components (all vertices) written in geometry stage

maxFragmentInputComponents
max num of input compontents read in fragment stage

maxFragmentOutputAttachments
max num of output attachments written in fragment stage

maxFragmentDualSourceAttachments
max num of output attachments written when using dual source blending

maxFragmentCombinedOutputResources
    total num of storage buffers, storage images and output buffers

maxComputeSharedMemorySize
    max total storage size of work group local storage (bytes)

maxComputeWorkGroupCount[3]
    max num of compute work groups that may be dispatched by a single command (x,y,z)

maxComputeWorkGroupInvocations
    max total compute invocations in a single local work group

maxComputeWorkGroupSize[3]
    max local size of a compute work group (x,y,z)

subPixelPrecisionBits
    num bits of subpixel precision in screen x and y

subTexelPrecisionBits
    num bits of subtexel precision

mipmapPrecisionBits
    num bits of mipmap precision

maxDrawIndexedIndexValue
    max index value for indexed draw calls (for 32-bit indices)

maxDrawIndirectCount
    max draw count for indirect draw calls

maxSamplerLodBias
    max absolute sampler level of detail bias

maxSamplerAnisotropy
    max degree of sampler anisotropy

maxViewports
    max number of active viewports

maxViewportDimensions[2]
    max viewport dimensions (x,y)

viewportBoundsRange[2]
    viewport bounds range (min,max)

viewportSubPixelBits
    num bits of subpixel precision for viewport

minMemoryMapAlignment
    min required alignment of host-visible memory allocations within the host address space (bytes)

minTexelBufferOffsetAlignment
    min required alignment for texel buffer offsets (bytes)

minUniformBufferOffsetAlignment
    min required alignment for uniform buffer sizes and offsets (bytes)

minStorageBufferOffsetAlignment
    min required alignment for storage buffer offsets (bytes)

minTexelOffset
    min texel offset for OpTextureSampleOffset

maxTexelOffset
    max texel offset for OpTextureSampleOffset

minTexelGatherOffset
min texel offset for OpTextureGatherOffset

maxTexelGatherOffset
max texel offset for OpTextureGatherOffset

minInterpolationOffset
furthest negative offset for interpolateAtOffset

maxInterpolationOffset
furthest positive offset for interpolateAtOffset

subPixelInterpolationOffsetBits
num of subpixel bits for interpolateAtOffset

maxFramebufferWidth
max width for a framebuffer

maxFramebufferHeight
max height for a framebuffer

maxFramebufferLayers
max layer count for a layered framebuffer

framebufferColorSampleCounts
supported color sample counts for a framebuffer

framebufferDepthSampleCounts
supported depth sample counts for a framebuffer

framebufferStencilSampleCounts
supported stencil sample counts for a framebuffer

framebufferNoAttachmentsSampleCounts
supported sample counts for a framebuffer with no attachments

maxColorAttachments
max num of color attachments per subpass

sampledImageColorSampleCounts
supported sample counts for an image with a non-integer color format

sampledImageIntegerSampleCounts
supported sample counts for an image with an integer color format

sampledImageDepthSampleCounts
supported sample counts for an image with a depth format

sampledImageStencilSampleCounts
supported sample counts for an image with a stencil format

storageImageSampleCounts
supported sample counts for an image used for storage operations

timestampFrequency
1/clock_tick_granularity for timestamp queries

maxClipDistances
max number of clip distances

maxCullDistances
max number of cull distances

maxCombinedClipAndCullDistances
max combined number of user clipping

pointSizeRange[2]
range (min,max) of supported point sizes

lineWidthRange[2]
range (min,max) of supported line widths

pointSizeGranularity
granularity of supported point sizes

lineWidthGranularity
granularity of supported line widths

4.7.4 Description

4.7.5 See Also

`vkGetPhysicalDeviceFeatures`

4.8 VkPipelineLayoutCreateInfo(3)

4.8.1 Name

VkPipelineLayoutCreateInfo - Structure specifying the parameters of a newly created pipeline layout object.

4.8.2 C Specification

```
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkPipelineLayoutCreateFlags     flags;
    uint32_t                        setLayoutCount;
    const VkDescriptorSetLayout*    pSetLayouts;
    uint32_t                        pushConstantRangeCount;
    const VkPushConstantRange*      pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

4.8.3 Fields

sType
Structure type. Must be `VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO`.

pNext
Pointer to next structure in the structure chain when applicable.

setLayoutCount
Number of descriptor sets interfaced by the pipeline.

pSetLayouts
Pointer to an array of *`setLayoutCount`* number of descriptor set layout objects defining the layout of the descriptor set at the corresponding index.

4.8.4 Description

This structure is used to specify the parameters of pipeline layout objects created using `vkCreatePipelineLayout`.

Valid Usage

- *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
- *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*

4.8.5 See Also

vkCreatePipelineLayout

4.9 VkQueueFamilyProperties(3)

4.9.1 Name

VkQueueFamilyProperties - Structure providing information about a queue family.

4.9.2 C Specification

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

4.9.3 Fields

queueFlags
Capabilities of the queues in this queue family (see `VkQueueFlags` for more detail).

queueCount
Number of queues in this queue family.

supportsTimestamps
Tells whether queues in this queue family support timestamps.

4.9.4 Description

The properties of queue families available in this structure can be retrieved using `vkGetPhysicalDeviceQueueFamilyProperties`.

4.9.5 See Also

`vkGetPhysicalDeviceQueueFamilyProperties`, `VkQueueFlags`

4.10 VkWriteDescriptorSet(3)

4.10.1 Name

VkWriteDescriptorSet - Structure specifying the parameters of a descriptor set write operation.

4.10.2 C Specification

```
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;
```

4.10.3 Fields

sType
: Structure type. Must be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET`.

pNext
: Pointer to next structure in the structure chain when applicable.

dstSet
: Destination descriptor set to write the descriptor data to.

dstBinding
: Binding within the descriptor set to start the update from.

dstArrayElement
: Array element of the binding to start the update from.

descriptorCount
: Number of descriptors to write to the descriptor set.

descriptorType
: Type of descriptors to write to the descriptor set.

pImageInfo
: A pointer to an array of *descriptorCount* `VkDescriptorImageInfo` structures specifying the source of the descriptor data to write to the descriptor set for images.

pBufferInfo
: A pointer to an array of *descriptorCount* `VkDescriptorBufferInfo` structures specifying the source of the descriptor data to write to the descriptor set for buffers.

pTexelBufferView
: A pointer to an array of `VkBufferView` handles used when binding texel buffers into a the descriptor set.

4.10.4 Description

This structure specifies information about the descriptors to be written to a descriptor set using the **vkUpdateDescriptorSets** command.

When writing data to descriptor sets, the *pImageInfo*, *pBufferInfo* or *pTexelBufferView* parameters of **vkUpdateDescriptorSets** point to *descriptorCount* instances of data structures, each instance specifying the source of the descriptor data to be written. Which of these parameters is used depends on the value of *descriptorType*.

Each instance of the selected array allows writing *descriptorCount* descriptors of type *descriptorType* to the destination descriptor set specified by *dstSet* starting from the array element index *dstArrayElement* of the *dstBinding* binding.

If *descriptorCount* is greater than the number of descriptors in the specified binding starting from the specified array element index then subsequent descriptors are written to the next binding starting from its first array element. This allows updating multiple subsequent bindings with a single instance of this structure as long as the descriptor type of those bindings match.

Attempting to write descriptors of incompatible type to any binding of a descriptor set may result in undefined behavior.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET
- *pNext* must be NULL
- *dstSet* must be a valid VkDescriptorSet handle
- *descriptorType* must be a valid VkDescriptorType value
- *descriptorCount* must be greater than 0
- Each of *dstSet* and the elements of *pTexelBufferView* that are valid handles must have been created, allocated or retrieved from the same VkDevice
- *dstBinding* must be a valid binding point within *dstSet*
- *descriptorType* must match the type of *dstBinding* 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 [?]
- 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 created 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 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* 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* must be less than or equal to VkPhysicalDeviceLimits::*maxStorageBufferRange*
- If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, the VkBuffer that any given ele-

4.10.5 See Also

vkUpdateDescriptorSets, VkDescriptorImageInfo, VkDescriptorBufferInfo, VkDescriptorType